目錄

keras-cn

本项目由BigMoyan于2016-4-29发起，旨在建立一个keras.io的中文版文档，并提供更多用户友好的支持与建议。

本项目目前已完成2.x版本，文档网址为keras-cn

如果你在使用keras和keras-cn的过程中有任何问题，请发在issue中，我会定时检查和回复，在issue中你也可以和志同道合的好基友们一起聊天~

项目基于Mkdocs生成静态网页，如果你想为文档做出贡献，请使用Markdown编写文档并遵守以下约定。

0.字体颜色

保持默认的字体颜色和字号，锚点的颜色为默认，超链接的颜色为默认

当使用 来设置锚点时，可能引起字体颜色改变，如果颜色发生改变，需要使用font修正字体，默认字体的颜色是#404040

1.标题级别

页面大标题为一级，一般每个文件只有一个一级标题 # 页面内的小节是二级标题 ## 小节的小节，如examples里各个example是三级标题 ###

2.代码块规则

成块的代码使用

```python

code

```

的形式显式指明 段中代码使用```code```的形式指明

3. 超链接

链接到本项目其他页面的超链接使用相对路径，一个例子是 `[<font color='#FF0000'>text</font>](../models/about_model.md)` 链接到其他外站的链接形式与此相同，只不过圆括号中是绝对地址

4.图片

图片保存在docs/images中，插入的例子是：

```
![text](../images/image_name.png)
```

5.分割线

每个二级标题之间使用 *** 产生一个分割线

参考网站

Markdown简明教程

Markdown

MkDocs中文教程

MkDocs

Keras文档

Keras

感谢参与！

Keras:基于Theano和TensorFlow的深度学习库

这就是Keras

Keras是一个高层神经网络API，Keras由纯Python编写而成并基Tensorflow或Theano。Keras 为支持快速实验而生，能够把你的idea迅速转换为结果，如果你有如下需求，请选择Keras：

- 简易和快速的原型设计（keras具有高度模块化，极简，和可扩充特性）
- 支持CNN和RNN，或二者的结合
- 无缝CPU和GPU切换

Keras适用的Python版本是：Python 2.7-3.5

Keras的设计原则是

- 用户友好：Keras是为人类而不是天顶星人设计的API。用户的使用体验始终是我们考虑的首要和中心内容。Keras遵循减少认知困难的最佳实践：Keras提供一致而简洁的API， 能够极大减少一般应用下用户的工作量，同时，Keras提供清晰和具有实践意义的bug反馈。

- 模块性：模型可理解为一个层的序列或数据的运算图，完全可配置的模块可以用最少的代价自由组合在一起。具体而言，网络层、损失函数、优化器、初始化策略、激活函数、正则化方法都是独立的模块，你可以使用它们来构建自己的模型。

- 易扩展性：添加新模块超级容易，只需要仿照现有的模块编写新的类或函数即可。创建新模块的便利性使得Keras更适合于先进的研究工作。

- 与Python协作：Keras没有单独的模型配置文件类型（作为对比，caffe有），模型由python代码描述，使其更紧凑和更易debug，并提供了扩展的便利性。

关于Keras-cn

本文档是Keras文档的中文版，包括keras.io的全部内容，以及更多的例子、解释和建议

现在，keras-cn的版本号将简单的跟随最新的keras release版本

由于作者水平和研究方向所限，无法对所有模块都非常精通，因此文档中不可避免的会出现各种错误、疏漏和不足之处。如果您在使用过程中有任何意见、建议和疑问，欢迎发送邮件到moyan_work@foxmail.com与我取得联系。

您对文档的任何贡献，包括文档的翻译、查缺补漏、概念解释、发现和修改问题、贡献示例程序等，均会被记录在致谢，十分感谢您对Keras中文文档的贡献！

如果你发现本文档缺失了官方文档的部分内容，请积极联系我补充。

本文档相对于原文档有更多的使用指导和概念澄清，请在使用时关注文档中的Tips，特别的，本文档的额外模块还有：

- Keras新手指南：我们新提供了“Keras新手指南”的页面，在这里我们对Keras进行了感性介绍，并简单介绍了Keras配置方法、一些小知识与使用陷阱，新手在使用前应该先阅读本部分的文档。
- Keras资源：在这个页面，我们罗列一些Keras可用的资源，本页面会不定期更新，请注意关注
- 深度学习与Keras：位于导航栏最下方的该模块翻译了来自Keras作者博客keras.io 和其他Keras相关博客的文章，该栏目的文章提供了对深度学习的理解和大量使用Keras的例子，您也可以向这个栏目投稿。 所有的文章均在醒目位置标志标明来源与作者，本文档对该栏目文章的原文不具有任何处置权。如您仍觉不妥，请联系本人（moyan_work@foxmail.com）删除。

当前版本与更新

如果你发现本文档提供的信息有误，有两种可能：

- 你的Keras版本过低：记住Keras是一个发展迅速的深度学习框架，请保持你的Keras与官方最新的release版本相符
- 我们的中文文档没有及时更新：如果是这种情况，请发邮件给我，我会尽快更新

目前文档的版本号是2.0.4，对应于官方的2.0.4 release 版本, 本次更新的主要内容是：

- 文档全面升级为Keras2，绝大多数API均得到更新，请及时查看。对于Keras 1的用户，请从github下载本文档源代码，在legacy中将Markdown文件生成为pdf查看，同时如果哪位同学做了keras1.2.2文档的pdf版也请发邮件告诉我
- 新增模块“keras新手指南”，敬请关注
- 补充了原文档中缺失，但源代码中确实可用的类和函数
- 重新整理了文档风格和栏目，使得文档更容易阅读，改变了一些词的译法
- 目前，“深度学习与Keras”一节中的示例代码尚未调整到Keras 2版本，敬请注意。
- 如果想查看Keras1和Keras2的API变动，请访问这里

注意，keras在github上的master往往要高于当前的release版本，如果你从源码编译keras，可能某些模块与文档说明不相符，请以官方Github代码为准

快速开始：30s上手Keras

Keras的核心数据结构是“模型”，模型是一种组织网络层的方式。Keras中主要的模型是Sequential模型，Sequential是一系列网络层按顺序构成的栈。你也可以查看函数式模型来学习建立更复杂的模型

Sequential模型如下

```
from keras.models import Sequential

model = Sequential()
```

将一些网络层通过 `.add()` 堆叠起来，就构成了一个模型：

```
from keras.layers import Dense, Activation

model.add(Dense(units=64, input_dim=100))
model.add(Activation("relu"))
model.add(Dense(units=10))
model.add(Activation("softmax"))
```

完成模型的搭建后，我们需要使用 `.compile()` 方法来编译模型：

```
model.compile(loss='categorical_crossentropy', optimizer='sgd',
metrics=['accuracy'])
```

编译模型时必须指明损失函数和优化器，如果你需要的话，也可以自己定制损失函数。Keras的一个核心理念就是简明易用同时，保证用户对Keras的绝对控制力度，用户可以根据自己的需要定制自己的模型、网络层，甚至修改源代码。

```
from keras.optimizers import SGD
model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=
0.01, momentum=0.9, nesterov=True))
```

完成模型编译后，我们在训练数据上按batch进行一定次数的迭代来训练网络

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

当然，我们也可以手动将一个个batch的数据送入网络中训练，这时候需要使用：

```
model.train_on_batch(x_batch, y_batch)
```

随后，我们可以使用一行代码对我们的模型进行评估，看看模型的指标是否满足我们的要求：

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128
)
```

或者，我们可以使用我们的模型，对新的数据进行预测：

```
classes = model.predict(x_test, batch_size=128)
```

搭建一个问答系统、图像分类模型，或神经图灵机、word2vec词嵌入器就是这么快。支撑深度学习的基本想法本就是简单的，现在让我们把它的实现也变的简单起来！

为了更深入的了解Keras，我们建议你查看一下下面的两个tutorial

- 快速开始Sequntial模型
- 快速开始函数式模型

还有我们的新手教程，虽然是面向新手的，但我们阅读它们总是有益的：

- Keras新手指南

在Keras代码包的examples文件夹里，我们提供了一些更高级的模型：基于记忆网络的问答系统、基于LSTM的文本的文本生成等。

安装

Keras使用了下面的依赖包：

- numpy，scipy
- pyyaml
- HDF5, h5py（可选，仅在模型的save/load函数中使用）
- 如果使用CNN的推荐安装cuDNN

当使用TensorFlow为后端时：

- TensorFlow

当使用Theano作为后端时：

- Theano

“后端”翻译自backend，指的是Keras依赖于完成底层的张量运算的软件包。

安装Keras时，请 `cd` 到Keras的文件夹中，并运行下面的安装命令：

```
sudo python setup.py install
```

你也可以使用PyPI来安装Keras

```
sudo pip install keras
```

详细的**Windows**和**Linux**安装教程请参考**“Keras**新手指南**”**中给出的安装教程，特别鸣谢**SCP-173**编写了这些教程

在Theano和TensorFlow间切换

Keras默认使用TensorFlow作为后端来进行张量操作，如需切换到Theano，请查看这里

技术支持

你可以在下列网址提问或加入Keras开发讨论:

- Keras Google group
- Keras Slack channel,点击这里获得邀请.

你也可以在Github issues里提问或请求新特性。在提问之前请确保你阅读过我们的指导

另外，对于习惯中文的用户，我们推荐在“集智”平台提问，该平台由Kaiser等搭建，支持在线代码运行环境，我本人会经常访问该网站解答问题

最后，我们也欢迎同学们加我们的QQ群119427073进行讨论（潜水和灌水会被T，入群说明公司/学校-职位/年级）

小额赞助

如果你觉得本文档对你的研究和使用有所帮助，欢迎扫下面的二维码对作者进行小额赞助，以鼓励作者进一步完善文档内容，提高文档质量。同时，不妨为本文档的github加颗星哦



keras新手指南

一些基本概念

在开始学习Keras之前，我们希望传递一些关于Keras，关于深度学习的基本概念和技术，我们建议新手在使用Keras之前浏览一下本页面提到的内容，这将减少你学习中的困惑

符号计算

Keras的底层库使用Theano或TensorFlow，这两个库也称为Keras的后端。无论是Theano还是TensorFlow，都是一个“符号式”的库。

因此，这也使得Keras的编程与传统的Python代码有所差别。笼统的说，符号主义的计算首先定义各种变量，然后建立一个“计算图”，计算图规定了各个变量之间的计算关系。建立好的计算图需要编译以确定其内部细节，然而，此时的计算图还是一个“空壳子”，里面没有任何实际的数据，只有当你把需要运算的输入放进去后，才能在整个模型中形成数据流，从而形成输出值。

就像用管道搭建供水系统，当你在拼水管的时候，里面是没有水的。只有所有的管子都接完了，才能送水。

Keras的模型搭建形式就是这种方法，在你搭建Keras模型完毕后，你的模型就是一个空壳子，只有实际生成可调用的函数后（K.function），输入数据，才会形成真正的数据流。

使用计算图的语言，如Theano，以难以调试而闻名，当Keras的Debug进入Theano这个层次时，往往也令人头痛。没有经验的开发者很难直观的感受到计算图到底在干些什么。尽管很让人头痛，但大多数的深度学习框架使用的都是符号计算这一套方法，因为符号计算能够提供关键的计算优化、自动求导等功能。

我们建议你在使用前稍微了解一下Theano或TensorFlow，Bing/Google一下即可。

张量

张量，或tensor，是本文档会经常出现的一个词汇，在此稍作解释。

使用这个词汇的目的是为了表述统一，张量可以看作是向量、矩阵的自然推广，我们用张量来表示广泛的数据类型。

规模最小的张量是0阶张量，即标量，也就是一个数。

当我们把一些数有序的排列起来，就形成了1阶张量，也就是一个向量

如果我们继续把一组向量有序的排列起来，就形成了2阶张量，也就是一个矩阵

把矩阵摞起来，就是3阶张量，我们可以称为一个立方体，具有3个颜色通道的彩色图片就是一个这样的立方体

把立方体摞起来，好吧这次我们真的没有给它起别名了，就叫4阶张量了，不要去试图想像4阶张量是什么样子，它就是个数学上的概念。

张量的阶数有时候也称为维度，或者轴，轴这个词翻译自英文axis。譬如一个矩阵[[1,2],[3,4]]，是一个2阶张量，有两个维度或轴，沿着第0个轴（为了与python的计数方式一致，本文档维度和轴从0算起）你看到的是[1,2]，[3,4]两个向量，沿着第1个轴你看到的是[1,3]，[2,4]两个向量。

要理解“沿着某个轴”是什么意思，不妨试着运行一下下面的代码：

```
import numpy as np

a = np.array([[1,2],[3,4]])
sum0 = np.sum(a, axis=0)
sum1 = np.sum(a, axis=1)

print sum0
print sum1
```

关于张量，目前知道这么多就足够了。事实上我也就知道这么多

data_format

这是一个无可奈何的问题，在如何表示一组彩色图片的问题上，Theano和TensorFlow发生了分歧，'th'模式，也即Theano模式会把100张RGB三通道的16×32（高为16宽为32）彩色图表示为下面这种形式（100,3,16,32），Caffe采取

的也是这种方式。第0个维度是样本维，代表样本的数目，第1个维度是通道维，代表颜色通道数。后面两个就是高和宽了。这种theano风格的数据组织方法，称为“channels_first”，即通道维靠前。

而TensorFlow，的表达形式是（100,16,32,3），即把通道维放在了最后，这种数据组织方式称为“channels_last”。

Keras默认的数据组织形式在~/.keras/keras.json中规定，可查看该文件的 `image_data_format` 一项查看，也可在代码中通过K.image_data_format()函数返回，请在网络的训练和测试中保持维度顺序一致。

唉，真是蛋疼，你们商量好不行吗？

函数式模型

函数式模型算是本文档比较原创的词汇了，所以这里要说一下

在Keras 0.x中，模型其实有两种，一种叫Sequential，称为序贯模型，也就是单输入单输出，一条路通到底，层与层之间只有相邻关系，跨层连接统统没有。这种模型编译速度快，操作上也比较简单。第二种模型称为Graph，即图模型，这个模型支持多输入多输出，层与层之间想怎么连怎么连，但是编译速度慢。可以看到，Sequential其实是Graph的一个特殊情况。

在Keras1和Keras2中，图模型被移除，而增加了了“functional model API”，这个东西，更加强调了Sequential是特殊情况这一点。一般的模型就称为Model，然后如果你要用简单的Sequential，OK，那还有一个快捷方式Sequential。

由于functional model API在使用时利用的是“函数式编程”的风格，我们这里将其译为函数式模型。总而言之，只要这个东西接收一个或一些张量作为输入，然后输出的也是一个或一些张量，那不管它是什么鬼，统统都称作“模型”。

batch

这个概念与Keras无关，老实讲不应该出现在这里的，但是因为它频繁出现，而且不了解这个技术的话看函数说明会很头痛，这里还是简单说一下。

深度学习的优化算法，说白了就是梯度下降。每次的参数更新有两种方式。

第一种，遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。这种方法每更新一次参数都要把数据集里的所有样本都看一遍，计算量开销大，计算速度慢，不支持在线学习，这称为Batch gradient descent，批梯度下降。

另一种，每看一个数据就算一下损失函数，然后求梯度更新参数，这个称为随机梯度下降，stochastic gradient descent。这个方法速度比较快，但是收敛性能不太好，可能在最优点附近晃来晃去，hit不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈。

为了克服两种方法的缺点，现在一般采用的是一种折中手段，mini-batch gradient decent，小批的梯度下降，这种方法把数据分为若干个批，按批来更新参数，这样，一个批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性。另一方面因为批的样本数与整个数据集相比小了很多，计算量也不是很大。

基本上现在的梯度下降都是基于mini-batch的，所以Keras的模块中经常会出现batch_size，就是指这个。

顺便说一句，Keras中用的优化器SGD是stochastic gradient descent的缩写，但不代表是一个样本就更新一回，还是基于mini-batch的。

epochs

真的不是很想解释这个词，但是新手问的还挺多的…… 简单说，epochs指的就是训练过程中数据将被“轮”多少次，就这样。

对新手友好的小说明

虽然这不是我们应该做的工作，但为了体现本教程对新手的友好，我们在这里简单列一下使用keras需要的先行知识。稍有经验的研究者或开发者请忽略本节，对于新手，我们建议在开始之前，确保你了解下面提到的术语的基本概念。如果你确实对某项内容不了解，请首先查阅相关资料，以免在未来使用中带来困惑。

关于Python

- 显然你应对Python有一定的熟悉，包括其基本语法，数据类型，语言特点等，如果你还不能使用Python进行程序设计，或不能避免Python中常见的一些小陷阱，或许你应该先去找个教程补充一下。这里推一个快速学习Python的教程廖雪峰的Python教程
- 你应该有面向对象的概念，知道类、对象、封装、多态、继承、作用域等术语的含义。
- 你应该对Python的科学计算包和深度学习包有一定了解，这些包包含但不限于numpy, scipy, scikit-learn, pandas...
- 特别地，你需要了解什么是生成器函数（generator），以及如何编写生成器函数。什么是匿名函数（lambda）

关于深度学习

由于Keras是为深度学习设计的工具，我们这里只列举深度学习中的一些基本概念。请确保你对下面的概念有一定理解。

- 有监督学习，无监督学习，分类，聚类，回归
- 神经元模型，多层感知器，BP算法
- 目标函数（损失函数），激活函数，梯度下降法
- 全连接网络、卷积神经网络、递归神经网络
- 训练集，测试集，交叉验证，欠拟合，过拟合
- 数据规范化
- 其他我还没想到的东西……想到再补充

其他

其他需要注意的概念，我们将使用[Tips]标注出来，如果该概念反复出现又比较重要，我们会写到这里。就酱，玩的愉快哟。

Keras FAQ：常见问题

如何引用Keras？

如果Keras对你的研究有帮助的话，请在你的文章中引用Keras。这里是一个使用BibTex的例子

```
@misc{chollet2015keras,
  author = {Chollet, François and others},
  title = {Keras},
  year = {2015},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {\url{https://github.com/fchollet/keras}}
}
```

如何使Keras调用GPU？

如果采用TensorFlow作为后端，当机器上有可用的GPU时，代码会自动调用GPU进行并行计算。如果使用Theano作为后端，可以通过以下方法设置：

方法1：使用Theano标记

在执行python脚本时使用下面的命令：

```
THEANO_FLAGS=device=gpu,floatX=float32 python my_keras_script.py
```

方法2：设置 `.theano` 文件

点击这里查看指导教程

方法3：在代码的开头处手动设置 `theano.config.device` 和 `theano.config.floatX`

```
    import theano
    theano.config.device = 'gpu'
    theano.config.floatX = 'float32'
```

如何保存Keras模型？

我们不推荐使用pickle或cPickle来保存Keras模型

你可以使用 `model.save(filepath)` 将Keras模型和权重保存在一个HDF5文件中，该文件将包含：

- 模型的结构，以便重构该模型
- 模型的权重
- 训练配置（损失函数，优化器等）
- 优化器的状态，以便于从上次训练中断的地方开始

使用 `keras.models.load_model(filepath)` 来重新实例化你的模型，如果文件中存储了训练配置的话，该函数还会同时完成模型的编译

例子：

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

如果你只是希望保存模型的结构，而不包含其权重或配置信息，可以使用：

```
# save as JSON
json_string = model.to_json()

# save as YAML
yaml_string = model.to_yaml()
```

这项操作将把模型序列化为json或yaml文件，这些文件对人而言也是友好的，如果需要的话你甚至可以手动打开这些文件并进行编辑。

当然，你也可以从保存好的json文件或yaml文件中载入模型：

```
# model reconstruction from JSON:
from keras.models import model_from_json
model = model_from_json(json_string)

# model reconstruction from YAML
model = model_from_yaml(yaml_string)
```

如果需要保存模型的权重，可通过下面的代码利用HDF5进行保存。注意，在使用前需要确保你已安装了HDF5和其Python库h5py

```
model.save_weights('my_model_weights.h5')
```

如果你需要在代码中初始化一个完全相同的模型，请使用：

```
model.load_weights('my_model_weights.h5')
```

如果你需要加载权重到不同的网络结构（有些层一样）中，例如fine-tune或transfer-learning，你可以通过层名字来加载模型：

```
model.load_weights('my_model_weights.h5', by_name=True)
```

例如：

```
"""
假如原模型为：
    model = Sequential()
    model.add(Dense(2, input_dim=3, name="dense_1"))
    model.add(Dense(3, name="dense_2"))
    ...
    model.save_weights(fname)
"""
# new model
model = Sequential()
model.add(Dense(2, input_dim=3, name="dense_1"))  # will be load
ed
model.add(Dense(10, name="new_dense"))  # will not be loaded

# load weights from first model; will only affect the first laye
r, dense_1.
model.load_weights(fname, by_name=True)
```

为什么训练误差比测试误差高很多？

一个Keras的模型有两个模式：训练模式和测试模式。一些正则机制，如Dropout，L1/L2正则项在测试模式下将不被启用。

另外，训练误差是训练数据每个batch的误差的平均。在训练过程中，每个epoch起始时的batch的误差要大一些，而后面的batch的误差要小一些。另一方面，每个epoch结束时计算的测试误差是由模型在epoch结束时的状态决定的，这时候的网络将产生较小的误差。

【Tips】可以通过定义回调函数将每个epoch的训练误差和测试误差并作图，如果训练误差曲线和测试误差曲线之间有很大的空隙，说明你的模型可能有过拟合的问题。当然，这个问题与Keras无关。

如何获取中间层的输出？

一种简单的方法是创建一个新的 `Model` ，使得它的输出是你想要的那个输出

```
from keras.models import Model

model = ...  # create the original model

layer_name = 'my_layer'
intermediate_layer_model = Model(input=model.input,
                                 output=model.get_layer(layer_na
me).output)
intermediate_output = intermediate_layer_model.predict(data
```

此外，我们也可以建立一个Keras的函数来达到这一目的：

```
from keras import backend as K

# with a Sequential model
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([X])[0]
```

当然，我们也可以直接编写Theano和TensorFlow的函数来完成这件事

注意，如果你的模型在训练和测试两种模式下不完全一致，例如你的模型中含有Dropout层，批规范化（BatchNormalization）层等组件，你需要在函数中传递一个learning_phase的标记，像这样：

```
get_3rd_layer_output = K.function([model.layers[0].input, K.lear
ning_phase()],
                                  [model.layers[3].output])

# output in test mode = 0
layer_output = get_3rd_layer_output([X, 0])[0]

# output in train mode = 1
layer_output = get_3rd_layer_output([X, 1])[0]
```

如何利用Keras处理超过机器内存的数据集？

可以使用 `model.train_on_batch(X,y)` 和 `model.test_on_batch(X,y)` 。请参考模型

另外，也可以编写一个每次产生一个batch样本的生成器函数，并调用 `model.fit_generator(data_generator, samples_per_epoch, nb_epoch)` 进行训练

这种方式在Keras代码包的example文件夹下CIFAR10例子里有示范，也可点击这里在github上浏览。

当验证集的loss不再下降时，如何中断训练？

可以定义 `EarlyStopping` 来提前终止训练

```
from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience=2)
model.fit(X, y, validation_split=0.2, callbacks=[early_stopping]
)
```

请参考回调函数

验证集是如何从训练集中分割出来的？

如果在 `model.fit` 中设置 `validation_spilt` 的值，则可将数据分为训练集和验证集，例如，设置该值为0.1，则训练集的最后10%数据将作为验证集，设置其他数字同理。注意，原数据在进行验证集分割前并没有被shuffle，所以这里的验证集

严格的就是你输入数据最末的x%。

训练数据在训练时会被随机洗乱吗？

是的，如果`model.fit`的`shuffle`参数为真，训练的数据就会被随机洗乱。不设置时默认为真。训练数据会在每个epoch的训练中都重新洗乱一次。

验证集的数据不会被洗乱

如何在每个**epoch**后记录训练/测试的**loss**和正确率？


```
```python
hist = model.fit(X, y, validation_split=0.2)
print(hist.history)
```

如何使用状态**RNN**（**statful RNN**）？

一个RNN是状态RNN，意味着训练时每个batch的状态都会被重用于初始化下一个batch的初始状态。

当使用状态RNN时，有如下假设

- 所有的batch都具有相同数目的样本
- 如果`X1`和`X2`是两个相邻的batch，那么对于任何`i`，`X2[i]`都是`X1[i]`的后续序列
```

要使用状态RNN，我们需要

- 显式的指定每个batch的大小。可以通过模型的首层参数`batch_input_shape`来完成。`batch_input_shape`是一个整数tuple，例如(32,10,16)代表一个具有10个时间步，每步向量长为16，每32个样本构成一个batch的输入数据格式。
- 在RNN层中，设置`stateful=True`

要重置网络的状态，使用：

- `model.reset_states()`来重置网络中所有层的状态
- `layer.reset_states()`来重置指定层的状态

例子：

```
X  # this is our input data, of shape (32, 21, 16)
# we will feed it to our model in sequences of length 10

model = Sequential()
model.add(LSTM(32, input_shape=(10, 16), batch_size=32, stateful=True))
model.add(Dense(16, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# we train the network to predict the 11th timestep given the first 10:
model.train_on_batch(X[:, :10, :], np.reshape(X[:, 10, :], (32, 16)))

# the state of the network has changed. We can feed the follow-up sequences:
model.train_on_batch(X[:, 10:20, :], np.reshape(X[:, 20, :], (32, 16)))

# let's reset the states of the LSTM layer:
model.reset_states()

# another way to do it in this case:
model.layers[0].reset_states()
```

注意，`predict`，`fit`，`train_on_batch`，`predict_classes`等方法都会更新模型中状态层的状态。这使得你可以不但可以进行状态网络的训练，也可以进行状态网络的预测。

如何“冻结”网络的层？

“冻结”一个层指的是该层将不参加网络训练，即该层的权重永不会更新。在进行fine-tune时我们经常会需要这项操作。 在使用固定的embedding层处理文本输入时，也需要这个技术。

可以通过向层的构造函数传递 `trainable` 参数来指定一个层是不是可训练的，如：

```
frozen_layer = Dense(32,trainable=False)
```

此外，也可以通过将层对象的 `trainable` 属性设为 `True` 或 `False` 来为已经搭建好的模型设置要冻结的层。 在设置完后，需要运行 `compile` 来使设置生效，例如：

```
x = Input(shape=(32,))
layer = Dense(32)
layer.trainable = False
y = layer(x)

frozen_model = Model(x, y)
# in the model below, the weights of `layer` will not be updated
 during training
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
# with this model the weights of the layer will be updated durin
g training
# (which will also affect the above model since it uses the same
 layer instance)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels)  # this does NOT update the weigh
ts of `layer`
trainable_model.fit(data, labels)  # this updates the weights of
 `layer`
```

如何从Sequential模型中去除一个层？

可以通过调用 `.pop()` 来去除模型的最后一个层，反复调用n次即可去除模型后面的n个层

```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(32, activation='relu'))

print(len(model.layers))  # "2"

model.pop()
print(len(model.layers))  # "1"
```

如何在Keras中使用预训练的模型？

我们提供了下面这些图像分类的模型代码及预训练权重：

- VGG16
- VGG19
- ResNet50
- Inception v3

可通过 `keras.applications` 载入这些模型：

```
from keras.applications.vgg16 import VGG16
from keras.applications.vgg19 import VGG19
from keras.applications.resnet50 import ResNet50
from keras.applications.inception_v3 import InceptionV3

model = VGG16(weights='imagenet', include_top=True)
```

这些代码的使用示例请参考 `.Application` 模型的文档

使用这些预训练模型进行特征抽取或fine-tune的例子可以参考此博客

VGG模型也是很多Keras例子的基础模型，如：

- Style-transfer
- Feature visualization
- Deep dream

如何在Keras中使用HDF5输入？

你可以使用keras.utils中的 `HDF5Matrix` 类来读取HDF5输入，参考这里

可以直接使用HDF5数据库，示例

```
import h5py
with h5py.File('input/file.hdf5', 'r') as f:
    X_data = f['X_data']
    model.predict(X_data)
```

Keras的配置文件存储在哪里？

所有的Keras数据默认存储在：

```
$HOME/.keras/
```

对windows用户而言， `$HOME` 应替换为 `%USERPROFILE%`

当Keras无法在上面的位置创建文件夹时（例如由于权限原因），备用的地址是 `/tmp/.keras/`

Keras配置文件为JSON格式的文件，保存在 `$HOME/.keras/keras.json` 。默认的配置文件长这样：

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

该文件包含下列字段：

- 默认的图像数据格式 `channels_last` 或 `channels_first`
- 用于防止除零错误的 `epsilon`
- 默认的浮点数类型
- 默认的后端

类似的，缓存的数据集文件，即由 `get_file()` 下载的文件，默认保存在 `$HOME/.keras/datasets/`

关于计算机的硬件配置说明

推荐配置

如果您是高校学生或者高级研究人员，并且实验室或者个人资金充沛，建议您采用如下配置：

- 主板：X99型号或Z170型号
- CPU: i7-5830K或i7-6700K 及其以上高级型号
- 内存：品牌内存，总容量32G以上，根据主板组成4通道或8通道
- SSD： 品牌固态硬盘，容量256G以上
- 显卡：NVIDIA GTX 1080ti、NVIDIA GTX TITAN、NVIDIA GTX 1080、NVIDIA GTX 1070、NVIDIA GTX 1060 (顺序为优先建议，并且建议同一显卡，可以根据主板插槽数量购买多块，例如X99型号主板最多可以采用×4的显卡)
- 电源：由主机机容量的确定，一般有显卡总容量后再加200W即可

最低配置

如果您是仅仅用于自学或代码调试，亦或是条件所限仅采用自己现有的设备进行开发，那么您的电脑至少满足以下几点：

- CPU：Intel第三代i5和i7以上系列产品或同性能AMD公司产品

- 内存：总容量4G以上

CPU说明

- 大多数CPU目前支持多核多线程，那么如果您采用CPU加速，就可以使用多线程运算。这方面的优势对于服务器CPU志强系列尤为关键

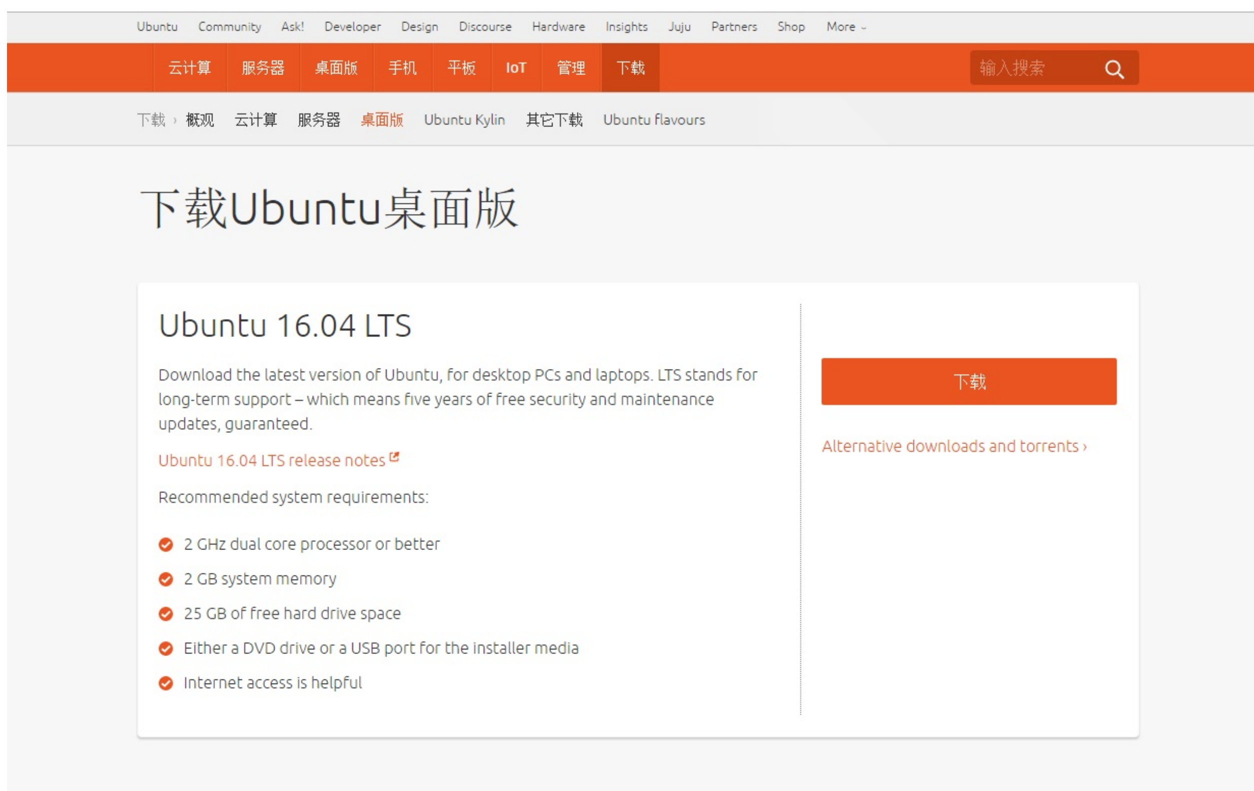
显卡说明

- 如果您的显卡是非NVIDIA公司的产品或是NVIDIA GTX系列中型号的第一个数字低于6或NVIDIA的GT系列，都不建议您采用此类显卡进行加速计算，例如 `NVIDIA GT 910` 、 `NVIDIA GTX 450` 等等。
- 如果您的显卡为笔记本上的GTX移动显卡（型号后面带有标识M），那么请您慎重使用显卡加速，因为移动版GPU容易发生过热烧毁现象。
- 如果您的显卡，显示的是诸如 `HD5000` , `ATI 5650` 等类型的显卡，那么您只能使用CPU加速
- 如果您的显卡芯片为Pascal架构（ `NVIDIA GTX 1080` , `NVIDIA GTX 1070` 等），您只能在之后的配置中选择 `CUDA 8.0`

基本开发环境搭建

1. Linux 发行版

linux有很多发行版，本文强烈建议读者采用新版的 `Ubuntu 16.04 LTS` 一方面，对于大多数新手来说Ubuntu具有很好的图形界面，与乐观的开源社区；另一方面，Ubuntu是Nvidia官方以及绝大多数深度学习框架默认开发环境。 个人不建议使用



通过U盘安装好后，进行初始化环境设置。

2. Ubuntu初始环境设置

- 安装开发包打开 `终端` 输入：

```
# 系统升级
>>> sudo apt update
>>> sudo apt upgrade
# 安装python基础开发包
>>> sudo apt install -y python-dev python-pip python-nose gc
c g++ git gfortran vim
```

- 安装运算加速库 打开 `终端` 输入：

```
>>> sudo apt install -y libopenblas-dev liblapack-dev libatl
as-base-dev
```

3. CUDA开发环境的搭建(CPU加速跳过)

3. CUDA开发环境的搭建(CPU加速跳过)

如果您的仅仅采用***cpu***加速，可跳过此步骤

- 下载CUDA8.0

下载地址：https://developer.nvidia.com/cuda-downloads

Select Target Platform
Click on the green buttons that describe your target platform. Only supported platforms will be shown.
Operating System: Windows | Linux | Mac OSX
Architecture: x86_64 | ppc64le
Distribution: Fedora | OpenSUSE | RHEL | CentOS | SLES | Ubuntu
Version: 16.04 | 14.04
Installer Type: runfile (local) | deb (local) | deb (network) | cluster (local)

Related Links
CUDA Quick Start Guide
Release Notes
EULA
Online Documentation
CUDA Toolkit Overview
Installer Checksums
Open Source Packages
Legacy CUDA Toolkits

Download Installer for Linux Ubuntu 16.04 x86_64
The base installer is available for download below.
Base Installer — Download (1.9 GB)
Installation Instructions:
1. `sudo dpkg -i cuda-repo-ubuntu1604-8-0-local_8.0.44-1_amd64.deb`
2. `sudo apt-get update`
3. `sudo apt-get install cuda`

The CUDA Toolkit contains Open-Source Software. The source code can be found here.
The checksums for the installer and patches can be found in Installer Checksums.
For further information, see the Installation Guide for Linux and the CUDA Quick Start Guide.

之后打开`终端`输入：

```
>>> sudo dpkg -i cuda-repo-ubuntu1604-8-0-local_8.0.44-1_amd64.d
eb
>>> sudo apt update
>>> sudo apt install cuda
```

自动配置成功就好。

- 将CUDA路径添加至环境变量 在`终端`输入：

  ```
  >>> sudo gedit /etc/bash.bashrc
  ```

```
export CUDA_HOME=/usr/local/cuda-8.0
export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

之后 `source gedit /etc/bash.bashrc` 即可 同样，在 `终端` 输入：

```
>>> sudo gedit ~/.bashrc
```

在 `.bashrc` 中添加如上相同内容（如果您使用的是 `zsh` ，在 `~/.zshrc` 添加即可）

- 测试 在 `终端` 输入：

```
>>> nvcc -V
```

会得到相应的nvcc编译器相应的信息，那么CUDA配置成功了。 记得重启系统

4. 加速库cuDNN（可选）

从官网下载需要注册账号申请，两三天批准。网盘搜索一般也能找到最新版。Linux目前就是cudnn-8.0-win-x64-v5.1-prod.zip。 下载解压出来是名为cuda的文件夹，里面有bin、include、lib，将三个文件夹复制到安装CUDA的地方覆盖对应文件夹，在终端中输入：

```
>>> sudo cp include/cudnn.h /usr/local/cuda-8.0/include/
>>> sudo cp lib64/* /usr/local/cuda-8.0/lib64/
```

Keras框架搭建

相关开发包安装

在 `终端` 中输入:

在 `终端` 中输入:

```
>>> sudo pip install -U --pre pip setuptools wheel
>>> sudo pip install -U --pre numpy scipy matplotlib scikit-lear
n scikit-image
>>> sudo pip install -U --pre tensorflow-gpu
# >>> sudo pip install -U --pre tensorflow
>>> sudo pip install -U --pre keras
```

安装完毕后，输入 `python` ，然后输入：

```
>>> import tensorflow
>>> import keras
```

无错输出即可

Keras中mnist数据集测试

下载Keras开发包

```
git clone https://github.com/fchollet/keras.git
cd keras/examples/
python mnist_mlp.py
```

程序无错进行，至此，keras安装完成。

声明与联系方式

由于作者水平和研究方向所限，无法对所有模块都非常精通，因此文档中不可避免的会出现各种错误、疏漏和不足之处。如果您在使用过程中有任何意见、建议和疑问，欢迎发送邮件到scp173.cool@gmail.com与作者取得联系.

本教程不得用于任何形式的商业用途，如果需要转载请与作者或中文文档作者联系，如果发现未经允许复制转载，将保留追求其法律责任的权利。

作者：SCP-173 E-mail ：scp173.cool@gmail.com 如果您需要及时得到指导帮助，可以加微信：**SCP173-cool**，酌情打赏即可



这里需要说明一下，笔者不建议在***Windows***环境下进行深度学习的研究，一方面是因为*Windows*所对应的框架搭建的依赖过多，社区设定不完全；另一方面，*Linux*系统下对显卡支持、内存释放以及存储空间调整等硬件功能支持较好。如果您对*Linux*环境感到陌生，并且大多数开发环境在*Windows*下更方便操作的话，希望这篇文章对您会有帮助。

由于**Keras**默认以**Tensorflow**为后端，且**Theano**后端更新缓慢，本文默认采用**Tensorflow1.0**作为**Keras**后端，**Theano**版安装方式请访问**[www.scp-173.top]**(http://www.scp-173.top)

关于计算机的硬件配置说明

推荐配置

如果您是高校学生或者高级研究人员，并且实验室或者个人资金充沛，建议您采用如下配置：

- 主板：X99型号、Z270型号
- CPU: i7-6850K或i7-7500K 及其以上高级型号
- 内存：品牌内存，总容量32G以上，根据主板组成4通道或8通道
- SSD： 品牌固态硬盘，容量256G以上
- 显卡：NVIDIA GTX 1080ti、NVIDIA GTX TITAN（Pascal）、NVIDIA GTX 1080、NVIDIA GTX 1070、NVIDIA GTX 1060 (顺序为优先性价比建议，并且建议同一显卡，可以根据主板插槽数量购买多块，例如X99型号主板最多可以采用×4的显卡)
- 电源：由主机机容量的确定，一般有显卡总容量后再加200W即可

最低配置

如果您是仅仅用于自学或代码调试，亦或是条件所限仅采用自己现有的设备进行开发，那么您的电脑至少满足以下几点：

- CPU：Intel第四代i5和i7以上系列产品或同性能AMD公司产品

- 显卡：计算能力高于NVIDIA GTX 750Ti的英伟达显卡

- 内存：总容量8G以上

CPU说明

- 大多数CPU目前支持多核多线程，那么如果您采用CPU加速，就可以使用多线程运算。这方面的优势对于服务器CPU Xeon系列尤为明显

显卡说明

- 如果您的显卡是非NVIDIA公司的产品或是NVIDIA GTX系列中型号的第一个数字低于4或NVIDIA的GT系列，都不建议您采用此类显卡进行加速计算，例如 `NVIDIA GT 910` 、 `NVIDIA GTX 450` 等等。
- 如果您的显卡为笔记本上的GTX移动显卡（型号后面带有标识M），那么请您慎重使用显卡加速，因为移动版GPU很容易发生过热烧毁现象。
- 如果您的显卡，显示的是诸如 `HD5000` , `ATI 5650` 等类型的显卡，那么您只能使用CPU加速

基本开发环境搭建

1. Microsoft Windows 版本

关于Windows的版本选择，本人强烈建议对于部分高性能的新机器采用 `Windows 10` 作为基础环境，部分老旧笔记本或低性能机器采用 `Windows 7` 即可，本文环境将以 `Windows 10` 作为开发环境进行描述。对于Windows 10的发行版本选择，笔者建议采用 `Windows_10_enterprise_2016_ltsb_x64` 作为基础环境。

这里推荐到MSDN我告诉你下载，也感谢作者国内优秀作者雪龙狼前辈所做出的贡献与牺牲。

直接贴出热链，复制粘贴迅雷下载：

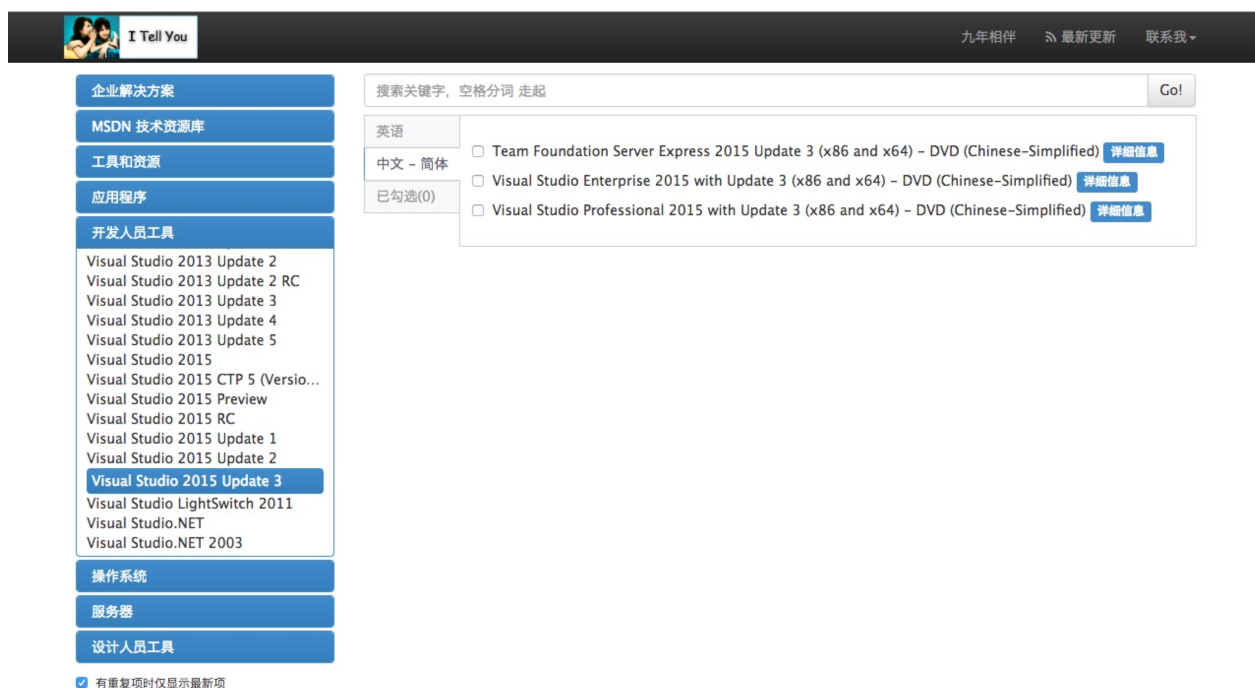
```
ed2k://|file|cn_windows_10_enterprise_2016_ltsb_x64_dvd_9060409.
iso|3821895680|FF17FF2D5919E3A560151BBC11C399D1|/
```

2. 编译环境Microsoft Visual Studio 2015 Update 3

(安装CPU版本非必须安装)

CUDA编译器为Microsoft Visual Studio，版本从2010-2015，其中 `cuda7.5` 仅支持2010、2012、2013， `cuda8.0` 仅支持2015版本，本文采用 `Visual Studio 2015 Update 3` 。同样直接贴出迅雷热链：

```
ed2k://|file|cn_visual_studio_professional_2015_with_update_3_x8
6_x64_dvd_8923256.iso|7745202176|DD35D3D169D553224BE5FB44E074ED5
E|/
```



3. Python环境

python环境建设推荐使用科学计算集成python发行版**Anaconda**，Anaconda是Python众多发行版中非常适用于科学计算的版本，里面已经集成了很多优秀的科学计算Python库。 建议安装 `Anconda3 4.2.0` 版本，目前新出的python3.6存在部分不兼容问题，所以建议安装历史版本4.2.0 注意：**windows**版本下的**tensorflow**暂时不支持**python2.7**

下载地址： Anaconda

4. CUDA

(安装CPU版本非必须安装) CUDA Toolkit是NVIDIA公司面向GPU编程提供的基础工具包，也是驱动显卡计算的核心技术工具。 直接安装CUDA8.0即可 下载地址：https://developer.nvidia.com/cuda-downloads

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System: Windows | Linux | Mac OSX

Architecture: x86_64

Version: 10 | 8.1 | 7 | Server 2012 R2 | Server 2008 R2

Installer Type: exe (network) | exe (local)

Related Links

CUDA Quick Start Guide
Release Notes
EULA
Online Documentation
CUDA Toolkit Overview
Installer Checksums
Open Source Packages
Legacy CUDA Toolkits

Download Installer for Windows 10 x86_64

The base installer is available for download below.

Base Installer — Download (1.2 GB)

Installation Instructions:

1. Double click cuda_8.0.44_win10.exe
2. Follow on-screen prompts

The checksums for the installer and patches can be found in Installer Checksums.
For further information, see the Installation Guide for Microsoft Windows and the CUDA Quick Start Guide.

在下载之后，按照步骤安装，不建议新手修改安装目录，同上，环境不需要配置，安装程序会自动配置好。

6. 加速库CuDNN

从官网下载需要注册 Nvidia 开发者账号，网盘搜索一般也能找到。 Windows目前最新版v6.0，但是keras尚未支持此版本，请下载v5.1版本，即 cudnn-8.0-win-x64-v5.1.zip。 下载解压出来是名为cuda的文件夹，里面有bin、include、lib，将三个文件夹复制到安装CUDA的地方覆盖对应文件夹，默认文件夹在： `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\8.0`

Keras 框架搭建

安装

在CMD命令行或者Powershell中输入：

```
# GPU 版本
>>> pip install --upgrade https://storage.googleapis.com/tensorf
low/windows/gpu/tensorflow_gpu-1.0.1-cp35-cp35m-win_amd64.whl

# CPU 版本
>>> pip install --upgrade https://storage.googleapis.com/tensorf
low/windows/cpu/tensorflow-1.0.1-cp35-cp35m-win_amd64.whl

# Keras 安装
>>> pip install keras -U --pre
```

之后可以验证keras是否安装成功,在命令行中输入Python命令进入Python变成命令行环境：

```
>>> import keras

Using Tensorflow backend.
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows
\tensorflow\stream_executor\dso_loader.cc:135] successfully open
ed CUDA library cublas64_80.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows
\tensorflow\stream_executor\dso_loader.cc:135] successfully open
ed CUDA library cudnn64_5.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows
\tensorflow\stream_executor\dso_loader.cc:135] successfully open
ed CUDA library cufft64_80.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows
\tensorflow\stream_executor\dso_loader.cc:135] successfully open
ed CUDA library nvcuda.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows
\tensorflow\stream_executor\dso_loader.cc:135] successfully open
ed CUDA library curand64_80.dll locally

>>>
```

没有报错，那么Keras就已经成功安装了

- Keras中mnist数据集测试 下载Keras开发包

```
>>> conda install git
>>> git clone https://github.com/fchollet/keras.git
>>> cd keras/examples/
>>> python mnist_mlp.py
```

程序无错进行，至此，keras安装完成。

Keras中文文档地址

声明与联系方式

由于作者水平和研究方向所限，无法对所有模块都非常精通，因此文档中不可避免的会出现各种错误、疏漏和不足之处。如果您在使用过程中有任何意见、建议和疑问，欢迎发送邮件到scp173.cool@gmail.com与中文文档作者取得联系.

本教程不得用于任何形式的商业用途，如果需要转载请与作者或中文文档作者联系，如果发现未经允许复制转载，将保留追求其法律责任的权利。

作者：SCP-173 E-mail ：scp173.cool@gmail.com 如果您需要及时得到指导帮助，可以加微信：**SCP173-cool**，酌情打赏即可



Keras使用陷阱

这里归纳了Keras使用过程中的一些常见陷阱和解决方法，如果你的模型怎么调都搞不对，或许你有必要看看是不是掉进了哪个猎人的陷阱，成为了一只嗷嗷待宰（？）的猎物

Keras陷阱不多，我们保持更新，希望能做一个陷阱大全

内有恶犬，小心哟

TF卷积核与TH卷积核

Keras提供了两套后端，Theano和Tensorflow，这是一件幸福的事，就像手中拿着馒头，想蘸红糖蘸红糖，想蘸白糖蘸白糖

如果你从无到有搭建自己的一套网络，则大可放心。但如果你想使用一个已有网络，或把一个用th/tf 训练的网络以另一种后端应用，在载入的时候你就应该特别小心了。

卷积核与所使用的后端不匹配，不会报任何错误，因为它们的shape是完全一致的，没有方法能够检测出这种错误。

在使用预训练模型时，一个建议是首先找一些测试样本，看看模型的表现是否与预计的一致。

如需对卷积核进行转换，可以使用utils.convert_all_kernels_in_model对模型的所有卷积核进行转换

向BN层中载入权重

如果你不知道从哪里淘来一个预训练好的BN层，想把它的权重载入到Keras中，要小心参数的载入顺序。

一个典型的例子是，将caffe的BN层参数载入Keras中，caffe的BN由两部分构成，bn层的参数是mean，std，scale层的参数是gamma，beta

按照BN的文章顺序，似乎载入Keras BN层的参数应该是[mean, std, gamma, beta]

然而不是的，Keras的BN层参数顺序应该是[gamma, beta, mean, std]，这是因为gamma和beta是可训练的参数，而mean和std不是

Keras的可训练参数在前，不可训练参数在后

错误的权重顺序不会引起任何报错，因为它们的shape完全相同

shuffle和validation_split的顺序

模型的fit函数有两个参数，shuffle用于将数据打乱，validation_split用于在没有提供验证集的时候，按一定比例从训练集中取出一部分作为验证集

这里有个陷阱是，程序是先执行validation_split，再执行shuffle的，所以会出现这种情况：

假如你的训练集是有序的，比方说正样本在前负样本在后，又设置了validation_split，那么你的验证集中很可能将全部是负样本

同样的，这个东西不会有任何错误报出来，因为Keras不可能知道你的数据有没有经过shuffle，保险起见如果你的数据是没shuffle过的，最好手动shuffle一下

Merge层的层对象与函数方法

Keras定义了一套用于融合张量的方法，位于keras.layers.Merge，里面有两套工具，以大写字母开头的是Keras Layer类，使用这种工具是需要实例化一个Layer对象，然后再使用。以小写字母开头的是张量函数方法，本质上是对Merge Layer对象的一个包装，但使用更加方便一些。注意辨析。

未完待续

如果你在使用Keras中遇到难以察觉的陷阱，请发信到moyan_work@foxmail.com说明~赠人玫瑰，手有余香，前人踩坑，后人沾光，有道是我不入地狱谁入地狱，愿各位Keras使用者积极贡献Keras陷阱。老规矩，陷阱贡献者将被列入致谢一栏

快速开始序贯（Sequential）模型

序贯模型是多个网络层的线性堆叠，也就是“一条路走到黑”。

可以通过向 `Sequential` 模型传递一个layer的list来构造该模型：

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
Dense(32, units=784),
Activation('relu'),
Dense(10),
Activation('softmax'),
])
```

也可以通过 `.add()` 方法一个个的将layer加入模型中：

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
model.add(Activation('relu'))
```

指定输入数据的shape

模型需要知道输入数据的shape，因此， `Sequential` 的第一层需要接受一个关于输入数据shape的参数，后面的各个层则可以自动的推导出中间数据的shape，因此不需要为每个层都指定这个参数。有几种方法来为第一层指定输入数据的shape

- 传递一个 `input_shape` 的关键字参数给第一层， `input_shape` 是一个tuple类型的数据，其中也可以填入 `None` ，如果填入 `None` 则表示此位置可能是任何正整数。数据的batch大小不应包含在其中。

- 有些2D层，如 `Dense` ，支持通过指定其输入维度 `input_dim` 来隐含的指定输入数据shape。一些3D的时域层支持通过参数 `input_dim` 和 `input_length` 来指定输入shape。

- 如果你需要为输入指定一个固定大小的batch_size（常用于stateful RNN网络），可以传递 `batch_size` 参数到一个层中，例如你想指定输入张量的batch大小是32，数据shape是（6，8），则你需要传递 `batch_size=32` 和 `input_shape=(6,8)` 。

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

```
model = Sequential()
model.add(Dense(32, input_shape=784))
```

编译

在训练模型之前，我们需要通过 `compile` 来对学习过程进行配置。 `compile` 接收三个参数：

- 优化器optimizer：该参数可指定为已预定义的优化器名，如 `rmsprop` 、 `adagrad` ，或一个 `Optimizer` 类的对象，详情见optimizers

- 损失函数loss：该参数为模型试图最小化的目标函数，它可为预定义的损失函数名，如 `categorical_crossentropy` 、 `mse` ，也可以为一个损失函数。详情见losses

- 指标列表metrics：对分类问题，我们一般将该列表设置为 `metrics=['accuracy']` 。指标可以是一个预定义指标的名字,也可以是一个用户定制的函数.指标函数应该返回单个张量,或一个完成 `metric_name - > metric_value` 映射的字典.请参考性能评估

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')

# For custom metrics
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

训练

Keras以Numpy数组作为输入数据和标签的数据类型。训练模型一般使用 `fit` 函数，该函数的详情见这里。下面是一些例子。

```
# For a single-input model with 2 classes (binary classification
):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# Train the model, iterating on the data in batches of 32 samples

model.fit(data, labels, epochs=10, batch_size=32)
```

```
# For a single-input model with 10 classes (categorical classifi
cation):

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=
10)

# Train the model, iterating on the data in batches of 32 samples

model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

例子

这里是一些帮助你开始的例子

在Keras代码包的examples文件夹中，你将找到使用真实数据的示例模型：

- CIFAR10 小图片分类：使用CNN和实时数据提升
- IMDB 电影评论观点分类：使用LSTM处理成序列的词语
- Reuters（路透社）新闻主题分类：使用多层感知器（MLP）
- MNIST手写数字识别：使用多层感知器和CNN
- 字符级文本生成：使用LSTM ...

基于多层感知器的softmax多分类：

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# Generate dummy data
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(
1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(
100, 1)), num_classes=10)

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data s
hape:
# here, 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

MLP的二分类：

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

# Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

类似VGG的卷积神经网络：

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD

# Generate dummy data
x_train = np.random.random((100, 100, 100, 3))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(
100, 1)), num_classes=10)
```

```
x_test = np.random.random((20, 100, 100, 3))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(
20, 1)), num_classes=10)

model = Sequential()
# input: 100x100 images with 3 channels -> (100, 100, 3) tensors.

# this applies 32 convolution filters of size 3x3 each.
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100
, 100, 3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

model.fit(x_train, y_train, batch_size=32, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=32)
```

使用LSTM的序列分类

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

使用1D卷积的序列分类

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPool
ing1D

model = Sequential()
model.add(Conv1D(64, 3, activation='relu', input_shape=(seq_leng
th, 100)))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(128, 3, activation='relu'))
model.add(Conv1D(128, 3, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

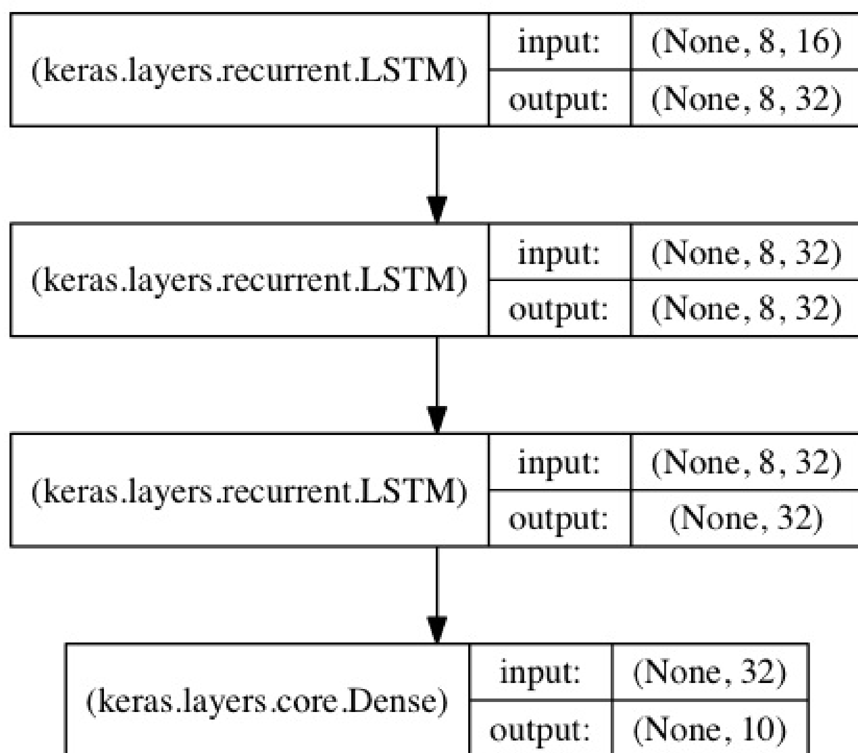
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

用于序列分类的栈式LSTM

在该模型中，我们将三个LSTM堆叠在一起，是该模型能够学习更高层次的时域特征表示。

开始的两层LSTM返回其全部输出序列，而第三层LSTM只返回其输出序列的最后一步结果，从而其时域维度降低（即将输入序列转换为单个向量）


(keras.layers.recurrent.LSTM)
input:
(None, 8, 16)
output:
(None, 8, 32)
(keras.layers.recurrent.LSTM)
input:
(None, 8, 32)
output:
(None, 8, 32)
(keras.layers.recurrent.LSTM)
input:
(None, 8, 32)
output:
(None, 32)
(keras.layers.core.Dense)
input:
(None, 32)
output:
(None, 10)

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10

# expected input data shape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  # returns a sequence of vectors of dimension 32
model.add(LSTM(32, return_sequences=True))  # returns a sequence of vectors of dimension 32
model.add(LSTM(32))  # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# Generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

# Generate dummy validation data
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))

model.fit(x_train, y_train,
          batch_size=64, epochs=5,
          validation_data=(x_val, y_val))
```

采用stateful LSTM的相同模型

stateful LSTM的特点是，在处理过一个batch的训练数据后，其内部状态（记忆）会被作为下一个batch的训练数据的初始状态。状态LSTM使得我们可以在合理的计算复杂度内处理较长序列

请FAQ中关于stateful LSTM的部分获取更多信息

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10
batch_size = 32

# Expected input batch shape: (batch_size, timesteps, data_dim)
# Note that we have to provide the full batch_input_shape since
the network is stateful.
# the sample of index i in batch k is the follow-up for the samp
le i in batch k-1.
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
               batch_input_shape=(batch_size, timesteps, data_di
m)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# Generate dummy training data
x_train = np.random.random((batch_size * 10, timesteps, data_dim
))
y_train = np.random.random((batch_size * 10, num_classes))

# Generate dummy validation data
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, num_classes))

model.fit(x_train, y_train,
          batch_size=batch_size, epochs=5, shuffle=False,
          validation_data=(x_val, y_val))
```

快速开始函数式（Functional）模型

我们起初将Functional一词译作泛型，想要表达该类模型能够表达任意张量映射的含义，但表达的不是很精确，在Keras 2里我们将这个词改译为“函数式”，对函数式编程有所了解的同学应能够快速get到该类模型想要表达的含义。函数式模型称作Functional，但它的类名是Model，因此我们有时候也用Model来代表函数式模型。

Keras函数式模型接口是用户定义多输出模型、非循环有向模型或具有共享层的模型等复杂模型的途径。一句话，只要你的模型不是类似VGG一样一条路走到黑的模型，或者你的模型需要多于一个的输出，那么你总应该选择函数式模型。函数式模型是最广泛的一类模型，序贯模型（Sequential）只是它的一种特殊情况。

这部分的文档假设你已经对Sequential模型已经比较熟悉

让我们从简单一点的模型开始

第一个模型：全连接网络

````
* 层对象接受张量为参数，返回一个张量。

* 输入是张量，输出也是张量的一个框架就是一个模型，通过```Model```定义。

* 这样的模型可以被像Keras的```Sequential```一样被训练

```python
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
````

所有的模型都是可调用的，就像层一样

利用函数式模型的接口，我们可以很容易的重用已经训练好的模型：你可以把模型当作一个层一样，通过提供一个tensor来调用它。注意当你调用一个模型时，你不仅仅重用了它的结构，也重用了它的权重。

```
x = Input(shape=(784,))
# This works, and returns the 10-way softmax we defined above.
y = model(x)
```
```

这种方式可以允许你快速的创建能处理序列信号的模型，你可以很快将一个图像分类的模型变为一个对视频分类的模型，只需要一行代码：

```
from keras.layers import TimeDistributed

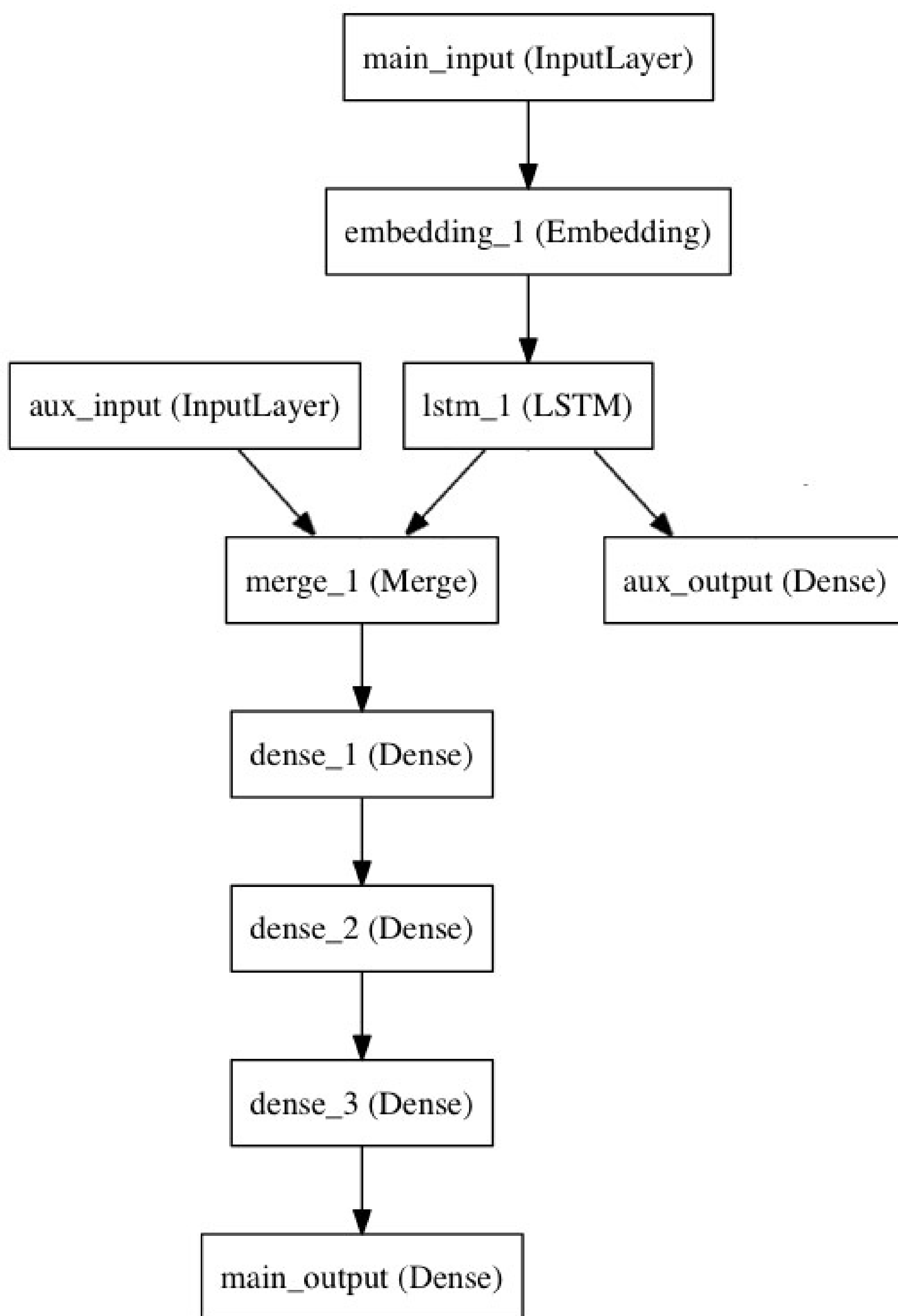
# Input tensor for sequences of 20 timesteps,
# each containing a 784-dimensional vector
input_sequences = Input(shape=(20, 784))

# This applies our previous model to every timestep in the input
 sequences.
# the output of the previous model was a 10-way softmax,
# so the output of the layer below will be a sequence of 20 vect
ors of size 10.
processed_sequences = TimeDistributed(model)(input_sequences)
```

多输入和多输出模型

使用函数式模型的一个典型场景是搭建多输入、多输出的模型。

考虑这样一个模型。我们希望预测Twitter上一条新闻会被转发和点赞多少次。模型的主要输入是新闻本身，也就是一个词语的序列。但我们还可以拥有额外的输入，如新闻发布的日期等。这个模型的损失函数将由两部分组成，辅助的损失函数评估仅仅基于新闻本身做出预测的情况，主损失函数评估基于新闻和额外信息的预测的情况，即使来自主损失函数的梯度发生弥散，来自辅助损失函数的信息也能够训练Embeddding和LSTM层。在模型中早点使用主要的损失函数是对于深度网络的一个良好的正则方法。总而言之，该模型框图如下：



让我们用函数式模型来实现这个框图

主要的输入接收新闻本身，即一个整数的序列（每个整数编码了一个词）。这些整数位于1到10，000之间（即我们的字典有10，000个词）。这个序列有100个单词。

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model

# Headline input: meant to receive sequences of 100 integers, be
tween 1 and 10000.
# Note that we can name any layer by passing it a "name" argumen
t.
main_input = Input(shape=(100,), dtype='int32', name='main_input'
)

# This embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)
(main_input)

# A LSTM will transform the vector sequence into a single vector,

# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

然后，我们插入一个额外的损失，使得即使在主损失很高的情况下，LSTM和Embedding层也可以平滑的训练。

```
auxiliary_output = Dense(1, activation='sigmoid', name='aux_outp
ut')(lstm_out)
```

再然后，我们将LSTM与额外的输入数据串联起来组成输入，送入模型中：

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])

# We stack a deep densely-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# And finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')
(x)
```

最后，我们定义整个2输入，2输出的模型：

```
model = Model(inputs=[main_input, auxiliary_input], outputs=[mai
n_output, auxiliary_output])
```

模型定义完毕，下一步编译模型。我们给额外的损失赋0.2的权重。我们可以通过关键字参数 `loss_weights` 或 `loss` 来为不同的输出设置不同的损失函数或权值。这两个参数均可为Python的列表或字典。这里我们给 `loss` 传递单个损失函数，这个损失函数会被应用于所有输出上。

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
              loss_weights=[1., 0.2])
```

编译完成后，我们通过传递训练数据和目标值训练该模型：

```
model.fit([headline_data, additional_data], [labels, labels],
          epochs=50, batch_size=32)
```

因为我们输入和输出是被命名过的（在定义时传递了“name”参数），我们也可以用下面的方式编译和训练模型：

```
model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output': 'binary_crossentropy'},
              loss_weights={'main_output': 1., 'aux_output': 0.2})

# And trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          epochs=50, batch_size=32)
```

共享层

另一个使用函数式模型的场合是使用共享层的时候。

考虑微博数据，我们希望建立模型来判别两条微博是否是来自同一个用户，这个需求同样可以用来判断一个用户的两条微博的相似性。

一种实现方式是，我们建立一个模型，它分别将两条微博的数据映射到两个特征向量上，然后将特征向量串联并加一个logistic回归层，输出它们来自同一个用户的概率。这种模型的训练数据是一对对的微博。

因为这个问题是对称的，所以处理第一条微博的模型当然也能重用于处理第二条微博。所以这里我们使用一个共享的LSTM层来进行映射。

首先，我们将微博的数据转为（140，256）的矩阵，即每条微博有140个字符，每个单词的特征由一个256维的词向量表示，向量的每个元素为1表示某个字符出现，为0表示不出现，这是一个one-hot编码。

之所以是（140，256）是因为一条微博最多有140个字符，而扩展的ASCII码表编码了常见的256个字符。原文中此处为Tweet，所以对外国人而言这是合理的。如果考虑中文字符，那一个单词的词向量就不止256了。

```
import keras
from keras.layers import Input, LSTM, Dense
from keras.models import Model

tweet_a = Input(shape=(140, 256))
tweet_b = Input(shape=(140, 256))
```

若要对不同的输入共享同一层，就初始化该层一次，然后多次调用它

```
# This layer can take as input a matrix
# and will return a vector of size 64
shared_lstm = LSTM(64)

# When we reuse the same layer instance
# multiple times, the weights of the layer
# are also being reused
# (it is effectively *the same* layer)
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# We can then concatenate the two vectors:
merged_vector = keras.layers.concatenate([encoded_a, encoded_b],
 axis=-1)

# And add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

# We define a trainable model linking the
# tweet inputs to the predictions
model = Model(inputs=[tweet_a, tweet_b], outputs=predictions)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit([data_a, data_b], labels, epochs=10)
```

先暂停一下，看看共享层到底输出了什么，它的输出数据shape又是什么

层“节点”的概念

无论何时，当你在某个输入上调用层时，你就创建了一个新的张量（即该层的输出），同时你也在为这个层增加一个“（计算）节点”。这个节点将输入张量映射为输出张量。当你多次调用该层时，这个层就有了多个节点，其下标分别为0，1，2...

在上一版本的Keras中，你可以通过 `layer.get_output()` 方法来获得层的输出张量，或者通过 `layer.output_shape` 获得其输出张量的shape。这个版本的Keras你仍然可以这么做（除了 `layer.get_output()` 被 `output` 替换）。但如果一个层与多个输入相连，会出现什么情况呢？

如果层只与一个输入相连，那没有任何困惑的地方。 `.output` 将会返回该层唯一的输出

```
a = Input(shape=(140, 256))

lstm = LSTM(32)
encoded_a = lstm(a)

assert lstm.output == encoded_a
```

但当层与多个输入相连时，会出现问题

```
a = Input(shape=(140, 256))
b = Input(shape=(140, 256))

lstm = LSTM(32)
encoded_a = lstm(a)
encoded_b = lstm(b)

lstm.output
```

上面这段代码会报错

```
>> AssertionError: Layer lstm_1 has multiple inbound nodes,
hence the notion of "layer output" is ill-defined.
Use `get_output_at(node_index)` instead.
```

通过下面这种调用方式即可解决

```
assert lstm.get_output_at(0) == encoded_a
assert lstm.get_output_at(1) == encoded_b
```


对于 `input_shape` 和 `output_shape` 也是一样，如果一个层只有一个节点，或所有的节点都有相同的输入或输出shape，那么 `input_shape` 和 `output_shape` 都是没有歧义的，并也只返回一个值。但是，例如你把一个相同的 `Conv2D` 应用于一个大小为(3,32,32)的数据，然后又将其应用于一个(3,64,64)的数据，那么此时该层就具有了多个输入和输出的shape，你就需要显式的指定节点的下标，来表明你想取的是哪个了

```
a = Input(shape=(3, 32, 32))
b = Input(shape=(3, 64, 64))

conv = Conv2D(16, (3, 3), padding='same')
conved_a = conv(a)

# Only one input so far, the following will work:
assert conv.input_shape == (None, 3, 32, 32)

conved_b = conv(b)
# now the `.input_shape` property wouldn't work, but this does:
assert conv.get_input_shape_at(0) == (None, 3, 32, 32)
assert conv.get_input_shape_at(1) == (None, 3, 64, 64)
```

更多的例子

代码示例依然是学习的最佳方式，这里是更多的例子

inception模型

inception的详细结构参见Google的这篇论文：Going Deeper with Convolutions

```
from keras.layers import Conv2D, MaxPooling2D, Input

input_img = Input(shape=(3, 256, 256))

tower_1 = Conv2D(64, (1, 1), padding='same', activation='relu')(
input_img)
tower_1 = Conv2D(64, (3, 3), padding='same', activation='relu')(
tower_1)

tower_2 = Conv2D(64, (1, 1), padding='same', activation='relu')(
input_img)
tower_2 = Conv2D(64, (5, 5), padding='same', activation='relu')(
tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), padding='same')(i
nput_img)
tower_3 = Conv2D(64, (1, 1), padding='same', activation='relu')(
tower_3)

output = keras.layers.concatenate([tower_1, tower_2, tower_3], a
xis=1)
```

卷积层的残差连接

残差网络（Residual Network）的详细信息请参考这篇文章：Deep Residual Learning for Image Recognition

```
from keras.layers import Conv2D, Input

# input tensor for a 3-channel 256x256 image
x = Input(shape=(3, 256, 256))
# 3x3 conv with 3 output channels (same as input channels)
y = Conv2D(3, (3, 3), padding='same')(x)
# this returns x + y.
z = keras.layers.add([x, y])
```

共享视觉模型

该模型在两个输入上重用了图像处理的模型，用来判别两个MNIST数字是否是相同的数字

```
from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Fla
tten
from keras.models import Model

# First, define the vision modules
digit_input = Input(shape=(1, 27, 27))
x = Conv2D(64, (3, 3))(digit_input)
x = Conv2D(64, (3, 3))(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)

vision_model = Model(digit_input, out)

# Then define the tell-digits-apart model
digit_a = Input(shape=(1, 27, 27))
digit_b = Input(shape=(1, 27, 27))

# The vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

concatenated = keras.layers.concatenate([out_a, out_b])
out = Dense(1, activation='sigmoid')(concatenated)

classification_model = Model([digit_a, digit_b], out)
```

视觉问答模型

在针对一幅图片使用自然语言进行提问时，该模型能够提供关于该图片的一个单词的答案

这个模型将自然语言的问题和图片分别映射为特征向量，将二者合并后训练一个logistic回归层，从一系列可能的回答中挑选一个。

```
from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense
from keras.models import Model, Sequential
```

```
# First, let's define a vision model using a Sequential model.
# This model will encode an image into a vector.
vision_model = Sequential()
vision_model.add(Conv2D(64, (3, 3) activation='relu', padding='same', input_shape=(3, 224, 224)))
vision_model.add(Conv2D(64, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(128, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

# Now let's get a tensor with the output of our vision model:
image_input = Input(shape=(3, 224, 224))
encoded_image = vision_model(image_input)

# Next, let's define a language model to encode the question into a vector.
# Each question will be at most 100 word long,
# and we will index words as integers from 1 to 9999.
question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256, input_length=100)(question_input)
encoded_question = LSTM(256)(embedded_question)

# Let's concatenate the question vector and the image vector:
merged = keras.layers.concatenate([encoded_question, encoded_image])

# And let's train a logistic regression over 1000 words on top:
output = Dense(1000, activation='softmax')(merged)

# This is our final model:
vqa_model = Model(inputs=[image_input, question_input], outputs=
```

```
output)

# The next stage would be training this model on actual data.
```

视频问答模型

在做完图片问答模型后，我们可以快速将其转为视频问答的模型。在适当的训练下，你可以为模型提供一个短视频（如100帧）然后向模型提问一个关于该视频的问题，如“what sport is the boy playing？”->“football”

```
from keras.layers import TimeDistributed

video_input = Input(shape=(100, 3, 224, 224))
# This is our video encoded via the previously trained vision_mo
del (weights are reused)
encoded_frame_sequence = TimeDistributed(vision_model)(video_inp
ut)  # the output will be a sequence of vectors
encoded_video = LSTM(256)(encoded_frame_sequence)  # the output
will be a vector

# This is a model-level representation of the question encoder,
reusing the same weights as before:
question_encoder = Model(inputs=question_input, outputs=encoded_
question)

# Let's use it to encode the question:
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

# And this is our video question answering model:
merged = keras.layers.concatenate([encoded_video, encoded_video_
question])
output = Dense(1000, activation='softmax')(merged)
video_qa_model = Model(inputs=[video_input, video_question_input
], outputs=output)
```

模型

关于Keras模型

Keras有两种类型的模型，序贯模型（Sequential）和函数式模型（Model），函数式模型应用更为广泛，序贯模型是函数式模型的一种特殊情况。

两类模型有一些方法是相同的：

- `model.summary()`：打印出模型概况
- `model.get_config()`:返回包含模型配置信息的Python字典。模型也可以从它的config信息中重构回去

```
config = model.get_config()
model = Model.from_config(config)
# or, for Sequential:
model = Sequential.from_config(config)
```

- `model.get_layer()`：依据层名或下标获得层对象
- `model.get_weights()`：返回模型权重张量的列表，类型为numpy array
- `model.set_weights()`：从numpy array里将权重载入给模型，要求数组具有与 `model.get_weights()` 相同的形状。
- `model.to_json`：返回代表模型的JSON字符串，仅包含网络结构，不包含权值。可以从JSON字符串中重构原模型：

```
from models import model_from_json

json_string = model.to_json()
model = model_from_json(json_string)
```

- `model.to_yaml`：与 `model.to_json` 类似，同样可以从产生的YAML字符串中重构模型

```
from models import model_from_yaml

yaml_string = model.to_yaml()
model = model_from_yaml(yaml_string)
```

- `model.save_weights(filepath)`：将模型权重保存到指定路径，文件类型是HDF5（后缀是.h5）
- `model.load_weights(filepath, by_name=False)`：从HDF5文件中加载权重到当前模型中, 默认情况下模型的结构将保持不变。如果想将权重载入不同的模型（有些层相同）中，则设置`by_name=True`，只有名字匹配的层才会载入权重

Sequential模型接口

如果刚开始学习Sequential模型，请首先移步这里阅读文档，本节内容是Sequential的API和参数介绍。

常用Sequential属性

- `model.layers` 是添加到模型上的层的list

Sequential模型方法

add

```
add(self, layer)
```

向模型中添加一个层

- layer: Layer对象

---

pop

```
pop(self)
```

弹出模型最后的一层，无返回值

---

compile

```
compile(self, optimizer, loss, metrics=None, sample_weight_mode=None)
```

编译用来配置模型的学习过程，其参数有

- optimizer：字符串（预定义优化器名）或优化器对象，参考优化器
- loss：字符串（预定义损失函数名）或目标函数，参考损失函数
- metrics：列表，包含评估模型在训练和测试时的网络性能的指标，典型用法是 `metrics=['accuracy']`
- sample_weight_mode：如果你需要按时间步为样本赋权（2D权矩阵），将该值设为“temporal”。默认为“None”，代表按样本赋权（1D权）。在下面 `fit` 函数的解释中有相关的参考内容。
- kwargs：使用TensorFlow作为后端请忽略该参数，若使用Theano作为后端，kwargs的值将会传递给 K.function

```
model = Sequential()
model.add(Dense(32, input_shape=(500,)))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
      loss='categorical_crossentropy',
      metrics=['accuracy'])
```

模型在使用前必须编译，否则在调用fit或evaluate时会抛出异常。

fit

```
fit(self, x, y, batch_size=32, epochs=10, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0)
```

本函数将模型训练 `nb_epoch` 轮，其参数有：

- x：输入数据。如果模型只有一个输入，那么x的类型是numpy array，如果模型有多个输入，那么x的类型应当为list，list的元素是对应于各个输入的numpy array
- y：标签，numpy array
- batch_size：整数，指定进行梯度下降时每个batch包含的样本数。训练时一个batch的样本会被计算一次梯度下降，使目标函数优化一步。
- epochs：整数，训练的轮数，每个epoch会把训练集轮一遍。
- verbose：日志显示，0为不在标准输出流输出日志信息，1为输出进度条记录，2为每个epoch输出一行记录
- callbacks：list，其中的元素是 `keras.callbacks.Callback` 的对象。这个list中的回调函数将会在训练过程中的适当时机被调用，参考回调函数
- validation_split：0~1之间的浮点数，用来指定训练集的一定比例数据作为验证集。验证集将不参与训练，并在每个epoch结束后测试的模型的指标，如损失函数、精确度等。注意，validation_split的划分在shuffle之前，因此如果你的数据本身是有序的，需要先手工打乱再指定validation_split，否则可能会出现验证集样本不均匀。
- validation_data：形式为（X，y）的tuple，是指定的验证集。此参数将覆盖validation_spilt。
- shuffle：布尔值或字符串，一般为布尔值，表示是否在训练过程中随机打乱输入样本的顺序。若为字符串“batch”，则是用来处理HDF5数据的特殊情况，它将在batch内部将数据打乱。
- class_weight：字典，将不同的类别映射为不同的权值，该参数用来在训练过程中调整损失函数（只能用于训练）
- sample_weight：权值的numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个1D的与样本等长的向量用于对样本进行1对1的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'` 。
- initial_epoch: 从该参数指定的epoch开始训练，在继续之前的训练时有用。

```
***
<a name='evaluate'>
<font color='#404040'>
### evaluate
</font>
</a>
```python
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

本函数按batch计算在某些输入数据上模型的误差，其参数有：

- x：输入数据，与 `fit` 一样，是numpy array或numpy array的list
- y：标签，numpy array
- batch_size：整数，含义同 `fit` 的同名参数
- verbose：含义同 `fit` 的同名参数，但只能取0或1
- sample_weight：numpy array，含义同 `fit` 的同名参数

本函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的list（如果模型还有其他的评价指标）。 `model.metrics_names` 将给出list中各个值的含义。

如果没有特殊说明，以下函数的参数均保持与 `fit` 的同名参数相同的含义

如果没有特殊说明，以下函数的verbose参数（如果有）均只能取0或1

---

predict

```
predict(self, x, batch_size=32, verbose=0)
```

本函数按batch获得输入数据对应的输出，其参数有：

函数的返回值是预测值的numpy array
```

predict_classes

```
predict_classes(self, x, batch_size=32, verbose=1)
```

本函数按batch产生输入数据的类别预测结果

函数的返回值是类别预测结果的numpy array或numpy

predict_proba

```
predict_proba(self, x, batch_size=32, verbose=1)
```

本函数按batch产生输入数据属于各个类别的概率

函数的返回值是类别概率的numpy array

train_on_batch

```
train_on_batch(self, x, y, class_weight=None, sample_weight=None
)
```

本函数在一个batch的数据上进行一次参数更新

函数返回训练误差的标量值或标量值的list，与evaluate的情形相同。

test_on_batch

```
test_on_batch(self, x, y, sample_weight=None)
```

本函数在一个batch的样本上对模型进行评估

函数的返回与evaluate的情形相同

predict_on_batch

```
predict_on_batch(self, x)
```

本函数在一个batch的样本上对模型进行测试

函数返回模型在一个batch上的预测结果

fit_generator

```
fit_generator(self, generator, steps_per_epoch, epochs=1, verbos
e=1, callbacks=None, validation_data=None, validation_steps=None
, class_weight=None, max_q_size=10, workers=1, pickle_safe=False
, initial_epoch=0)
```

利用Python的生成器，逐个生成数据的batch并进行训练。生成器与模型将并行执行以提高效率。例如，该函数允许我们在CPU上进行实时的数据提升，同时在GPU上进行模型训练

函数的参数是：

- generator：生成器函数，生成器的输出应该为：
    - 一个形如（inputs，targets）的tuple
    - 一个形如（inputs, targets,sample_weight）的tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个epoch以经过模型的样本数达到 `samples_per_epoch` 时，记一个epoch结束
- steps_per_epoch：整数，当生成器返回 `steps_per_epoch` 次数据时计一个epoch结束，执行下一个epoch

- epochs：整数，数据迭代的轮数
- verbose：日志显示，0为不在标准输出流输出日志信息，1为输出进度条记录，2为每个epoch输出一行记录
- validation_data：具有以下三种形式之一
    - 生成验证集的生成器
    - 一个形如（inputs,targets）的tuple
    - 一个形如（inputs,targets，sample_weights）的tuple
- validation_steps: 当validation_data为生成器时，本参数指定验证集的生成器返回次数
- class_weight：规定类别权重的字典，将类别映射为权重，常用于处理样本不均衡问题。
- sample_weight：权值的numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个1D的与样本等长的向量用于对样本进行1对1的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'` 。
- workers：最大进程数
- max_q_size：生成器队列的最大容量
- pickle_safe: 若为真，则使用基于进程的线程。由于该实现依赖多进程，不能传递non picklable（无法被pickle序列化）的参数到生成器中，因为无法轻易将它们传入子进程中。
- initial_epoch: 从该参数指定的epoch开始训练，在继续之前的训练时有用。

函数返回一个 `History` 对象

例子：

```
def generate_arrays_from_file(path):
    while 1:
            f = open(path)
            for line in f:
                # create Numpy arrays of input data
                # and labels, from each line in the file
                x, y = process_line(line)
                yield (x, y)
        f.close()

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
        samples_per_epoch=10000, epochs=10)
```

---

evaluate_generator

```
evaluate_generator(self, generator, steps, max_q_size=10, worker
s=1, pickle_safe=False)
```

本函数使用一个生成器作为数据源评估模型，生成器应返回与 `test_on_batch` 的输入数据相同类型的数据。该函数的参数与 `fit_generator` 同名参数含义相同，steps是生成器要返回数据的轮数。

---

predcit_generator

```
predict_generator(self, generator, steps, max_q_size=10, workers=
1, pickle_safe=False, verbose=0)
```

本函数使用一个生成器作为数据源预测模型，生成器应返回与 `test_on_batch` 的输入数据相同类型的数据。该函数的参数与 `fit_generator` 同名参数含义相同，steps是生成器要返回数据的轮数。

函数式模型接口

为什么叫“函数式模型”，请查看“Keras新手指南”的相关部分

Keras的函数式模型为 `Model` ，即广义的拥有输入和输出的模型，我们使用 `Model` 来初始化一个函数式模型

```
from keras.models import Model
from keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

在这里，我们的模型以 `a` 为输入，以 `b` 为输出，同样我们可以构造拥有多输入和多输出的模型

```
model = Model(inputs=[a1, a2], outputs=[b1, b3, b3])
```

常用Model属性

- `model.layers` ：组成模型图的各个层
- `model.inputs` ：模型的输入张量列表
- `model.outputs` ：模型的输出张量列表

Model模型方法

compile

```
compile(self, optimizer, loss, metrics=None, loss_weights=None,
sample_weight_mode=None)
```

本函数编译模型以供训练，参数有

- optimizer：优化器，为预定义优化器名或优化器对象，参考优化器
- loss：损失函数，为预定义损失函数名或一个目标函数，参考损失函数
- metrics：列表，包含评估模型在训练和测试时的性能的指标，典型用法是 `metrics=['accuracy']` 如果要在多输出模型中为不同的输出指定不同的指标，可像该参数传递一个字典，例如 `metrics={'ouput_a': 'accuracy'}`
- sample_weight_mode：如果你需要按时间步为样本赋权（2D权矩阵），将该值设为“temporal”。默认为“None”，代表按样本赋权（1D权）。如果模型有多个输出，可以向该参数传入指定sample_weight_mode的字典或列表。在下面 `fit` 函数的解释中有相关的参考内容。
- kwargs：使用TensorFlow作为后端请忽略该参数，若使用Theano作为后端，kwargs的值将会传递给 K.function

当为参数传入非法值时会抛出异常

【Tips】如果你只是载入模型并利用其predict，可以不用进行compile。在Keras中，compile主要完成损失函数和优化器的一些配置，是为训练服务的。predict会在内部进行符号函数的编译工作（通过调用_make_predict_function生成函数），

---

fit

```
fit(self, x=None, y=None, batch_size=32, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0)
```

本函数用以训练模型，参数有：

- x：输入数据。如果模型只有一个输入，那么x的类型是numpy array，如果模型有多个输入，那么x的类型应当为list，list的元素是对应于各个输入的numpy array。如果模型的每个输入都有名字，则可以传入一个字典，将输入名与其输入数据对应起来。

- y：标签，numpy array。如果模型有多个输出，可以传入一个numpy array的list。如果模型的输出拥有名字，则可以传入一个字典，将输出名与其标签对应起来。
- batch_size：整数，指定进行梯度下降时每个batch包含的样本数。训练时一个batch的样本会被计算一次梯度下降，使目标函数优化一步。
- nb_epoch：整数，训练的轮数，训练数据将会被遍历nb_epoch次。Keras中nb开头的变量均为"number of"的意思
- verbose：日志显示，0为不在标准输出流输出日志信息，1为输出进度条记录，2为每个epoch输出一行记录
- callbacks：list，其中的元素是 `keras.callbacks.Callback` 的对象。这个list中的回调函数将会在训练过程中的适当时机被调用，参考回调函数
- validation_split：0~1之间的浮点数，用来指定训练集的一定比例数据作为验证集。验证集将不参与训练，并在每个epoch结束后测试的模型的指标，如损失函数、精确度等。注意，validation_split的划分在shuffle之后，因此如果你的数据本身是有序的，需要先手工打乱再指定validation_split，否则可能会出现验证集样本不均匀。
- validation_data：形式为（X，y）或（X，y，sample_weights）的tuple，是指定的验证集。此参数将覆盖validation_spilt。
- shuffle：布尔值，表示是否在训练过程中每个epoch前随机打乱输入样本的顺序。
- class_weight：字典，将不同的类别映射为不同的权值，该参数用来在训练过程中调整损失函数（只能用于训练）。该参数在处理非平衡的训练数据（某些类的训练样本数很少）时，可以使得损失函数对样本数不足的数据更加关注。
- sample_weight：权值的numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个1D的与样本等长的向量用于对样本进行1对1的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'` 。
- initial_epoch: 从该参数指定的epoch开始训练，在继续之前的训练时有用。

输入数据与规定数据不匹配时会抛出错误

```
***

<a name='evaluate'>
<font color='#404040'>
### evaluate
</font>
</a>

```python
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=Non
e)
```

本函数按batch计算在某些输入数据上模型的误差，其参数有：

- x：输入数据，与 `fit` 一样，是numpy array或numpy array的list
- y：标签，numpy array
- batch_size：整数，含义同 `fit` 的同名参数
- verbose：含义同 `fit` 的同名参数，但只能取0或1
- sample_weight：numpy array，含义同 `fit` 的同名参数

本函数返回一个测试误差的标量值（如果模型没有其他评价指标），或一个标量的list（如果模型还有其他的评价指标）。 `model.metrics_names` 将给出list中各个值的含义。

如果没有特殊说明，以下函数的参数均保持与 `fit` 的同名参数相同的含义

如果没有特殊说明，以下函数的verbose参数（如果有）均只能取0或1

---

predict

```
predict(self, x, batch_size=32, verbose=0)
```

本函数按batch获得输入数据对应的输出，其参数有：
```

函数的返回值是预测值的numpy array

---

train_on_batch

```
train_on_batch(self, x, y, class_weight=None, sample_weight=None
)
```

本函数在一个batch的数据上进行一次参数更新

函数返回训练误差的标量值或标量值的list，与evaluate的情形相同。

---

test_on_batch

```
test_on_batch(self, x, y, sample_weight=None)
```

本函数在一个batch的样本上对模型进行评估

函数的返回与evaluate的情形相同

---

predict_on_batch

```
predict_on_batch(self, x)
```

本函数在一个batch的样本上对模型进行测试

函数返回模型在一个batch上的预测结果

---

fit_generator

```
fit_generator(self, generator, steps_per_epoch, epochs=1, verbos
e=1, callbacks=None, validation_data=None, validation_steps=None
, class_weight=None, max_q_size=10, workers=1, pickle_safe=False
, initial_epoch=0)
```

利用Python的生成器，逐个生成数据的batch并进行训练。生成器与模型将并行执行以提高效率。例如，该函数允许我们在CPU上进行实时的数据提升，同时在GPU上进行模型训练

函数的参数是：

- generator：生成器函数，生成器的输出应该为：
  - 一个形如（inputs，targets）的tuple
  - 一个形如（inputs, targets,sample_weight）的tuple。所有的返回值都应该包含相同数目的样本。生成器将无限在数据集上循环。每个epoch以经过模型的样本数达到 `samples_per_epoch` 时，记一个epoch结束
- steps_per_epoch：整数，当生成器返回 `steps_per_epoch` 次数据时计一个epoch结束，执行下一个epoch
- epochs：整数，数据迭代的轮数
- verbose：日志显示，0为不在标准输出流输出日志信息，1为输出进度条记录，2为每个epoch输出一行记录
- validation_data：具有以下三种形式之一
  - 生成验证集的生成器
  - 一个形如（inputs,targets）的tuple
  - 一个形如（inputs,targets，sample_weights）的tuple
- validation_steps: 当validation_data为生成器时，本参数指定验证集的生成器返回次数
- class_weight：规定类别权重的字典，将类别映射为权重，常用于处理样本不均衡问题。

- sample_weight：权值的numpy array，用于在训练时调整损失函数（仅用于训练）。可以传递一个1D的与样本等长的向量用于对样本进行1对1的加权，或者在面对时序数据时，传递一个的形式为（samples，sequence_length）的矩阵来为每个时间步上的样本赋不同的权。这种情况下请确定在编译模型时添加了 `sample_weight_mode='temporal'` 。
- workers：最大进程数
- max_q_size：生成器队列的最大容量
- pickle_safe: 若为真，则使用基于进程的线程。由于该实现依赖多进程，不能传递non picklable（无法被pickle序列化）的参数到生成器中，因为无法轻易将它们传入子进程中。
- initial_epoch: 从该参数指定的epoch开始训练，在继续之前的训练时有用。

函数返回一个 `History` 对象

例子

```
def generate_arrays_from_file(path):
    while 1:
    f = open(path)
    for line in f:
        # create numpy arrays of input data
        # and labels, from each line in the file
        x1, x2, y = process_line(line)
        yield ({'input_1': x1, 'input_2': x2}, {'output': y})
    f.close()

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
        steps_per_epoch=10000, epochs=10)
```

---

evaluate_generator

```
evaluate_generator(self, generator, steps, max_q_size=10, worker
s=1, pickle_safe=False)
```

本函数使用一个生成器作为数据源，来评估模型，生成器应返回与 `test_on_batch` 的输入数据相同类型的数据。

函数的参数是：

- generator：生成输入batch数据的生成器
- val_samples：生成器应该返回的总样本数
- steps：生成器要返回数据的轮数
- max_q_size：生成器队列的最大容量
- nb_worker：使用基于进程的多线程处理时的进程数
- pickle_safe：若设置为True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可pickle的参数到生成器中，因为它们不能轻易的传递到子进程中。

---

predict_generator

```
fit_generator(self, generator, steps_per_epoch, epochs=1, verbos
e=1, callbacks=None, validation_data=None, validation_steps=None
, class_weight=None, max_q_size=10, workers=1, pickle_safe=False
, initial_epoch=0)
```

从一个生成器上获取数据并进行预测，生成器应返回与 `predict_on_batch` 输入类似的数据

函数的参数是：

- generator：生成输入batch数据的生成器
- val_samples：生成器应该返回的总样本数
- max_q_size：生成器队列的最大容量
- nb_worker：使用基于进程的多线程处理时的进程数

- pickle_safe：若设置为True，则使用基于进程的线程。注意因为它的实现依赖于多进程处理，不可传递不可pickle的参数到生成器中，因为它们不能轻易的传递到子进程中。

网络层

关于Keras的“层”（Layer）

所有的Keras层对象都有如下方法：

- `layer.get_weights()` ：返回层的权重（numpy array）
- `layer.set_weights(weights)` ：从numpy array中将权重加载到该层中，要求numpy array的形状与* `layer.get_weights()` 的形状相同

  ```
  ```python
  layer = Dense(32)
  config = layer.get_config()
  reconstructed_layer = Dense.from_config(config)
  ```

或者：

```
from keras import layers

config = layer.get_config()
layer = layers.deserialize({'class_name': layer.__class__.__name
__,
                            'config': config})
```

如果层仅有一个计算节点（即该层不是共享层），则可以通过下列方法获得输入张量、输出张量、输入数据的形状和输出数据的形状：

- `layer.input`
- `layer.output`
- `layer.input_shape`
- `layer.output_shape`

如果该层有多个计算节点（参考层计算节点和共享层）。可以使用下面的方法

- `layer.get_input_at(node_index)`
- `layer.get_output_at(node_index)`
```

- `layer.get_input_shape_at(node_index)`
- `layer.get_output_shape_at(node_index)`

常用层

常用层对应于core模块，core内部定义了一系列常用的网络层，包括全连接、激活层等

Dense层

```
keras.layers.core.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

Dense就是常用的全连接层，所实现的运算是 `output = activation(dot(input, kernel)+bias)` 。其中 `activation` 是逐元素计算的激活函数， `kernel` 是本层的权值矩阵， `bias` 为偏置向量，只有当 `use_bias=True` 才会添加。

如果本层的输入数据的维度大于2，则会先被压为与 `kernel` 相匹配的大小。

这里是一个使用示例：

```
# as first layer in a sequential model:
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

参数：

- units：大于0的整数，代表该层的输出维度。

- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象

输入

形如(nb_samples, ..., input_shape[1])的nD张量，最常见的情况为(nb_samples, input_dim)的2D张量

输出

形如(nb_samples, ..., units)的nD张量，最常见的情况为(nb_samples, output_dim)的2D张量

Activation层


```
keras.layers.core.Activation(activation)
```

激活层对一个层的输出施加激活函数

参数

- activation：将要使用的激活函数，为预定义激活函数名或一个Tensorflow/Theano的函数。参考激活函数

输入shape

任意，当使用激活层作为第一层时，要指定 `input_shape`

输出shape

与输入shape相同

</a name='dropout'>

Dropout层


```
keras.layers.core.Dropout(rate, noise_shape=None, seed=None)
```

为输入数据施加Dropout。Dropout将在训练过程中每次更新参数时随机断开一定百分比（rate）的输入神经元，Dropout层用于防止过拟合。

参数

- rate：0~1的浮点数，控制需要断开的神经元的比例
- noise_shape：整数张量，为将要应用在输入上的二值Dropout mask的shape，例如你的输入为(batch_size, timesteps, features)，并且你希望在各个时间步上的Dropout mask都相同，则可传入noise_shape=(batch_size, 1, features)。

- seed：整数，使用的随机数种子

参考文献

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting

---

Flatten层

```
keras.layers.core.Flatten()
```

Flatten层用来将输入"压平"，即把多维的输入一维化，常用在从卷积层到全连接层的过渡。Flatten不影响batch的大小。

例子

```
model = Sequential()
model.add(Convolution2D(64, 3, 3,
            border_mode='same',
            input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

Reshape层

```
keras.layers.core.Reshape(target_shape)
```

Reshape层用来将输入shape转换为特定的shape

参数

- target_shape：目标shape，为整数的tuple，不包含样本数目的维度（batch大小）

输入shape

任意，但输入的shape必须固定。当使用该层为模型首层时，需要指定 `input_shape` 参数

输出shape

```
### 例子

```python
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

Permute层

```
keras.layers.core.Permute(dims)
```

Permute层将输入的维度按照给定模式进行重排，例如，当需要将RNN和CNN网络连接时，可能会用到该层。
```

参数

- dims：整数tuple，指定重排的模式，不包含样本数的维度。重拍模式的下标从1开始。例如（2，1）代表将输入的第二个维度重拍到输出的第一个维度，而将输入的第一个维度重排到第二个维度

例子

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
# now: model.output_shape == (None, 64, 10)
# note: `None` is the batch dimension
```

输入shape

任意，当使用激活层作为第一层时，要指定`input_shape`

输出shape

与输入相同，但是其维度按照指定的模式重新排列

RepeatVector层

```
keras.layers.core.RepeatVector(n)
```

RepeatVector层将输入重复n次

参数

- n：整数，重复的次数

输入shape

形如（nb_samples, features）的2D张量

输出shape

形如（nb_samples, n, features）的3D张量

例子

```
model = Sequential()
model.add(Dense(32, input_dim=32))
# now: model.output_shape == (None, 32)
# note: `None` is the batch dimension

model.add(RepeatVector(3))
# now: model.output_shape == (None, 3, 32)
```

Lambda层

```
keras.layers.core.Lambda(function, output_shape=None, mask=None,
 arguments=None)
```

本函数用以对上一层的输出施以任何Theano/TensorFlow表达式

参数

- function：要实现的函数，该函数仅接受一个变量，即上一层的输出
- output_shape：函数应该返回的值的shape，可以是一个tuple，也可以是一个根据输入shape计算输出shape的函数
- mask: 掩膜
- arguments：可选，字典，用来记录向函数中传递的其他关键字参数

例子

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
```

```
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part

def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2  # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,
         output_shape=antirectifier_output_shape))
```

输入shape

任意，当使用该层作为第一层时，要指定 `input_shape`

输出shape

由 `output_shape` 参数指定的输出shape，当使用tensorflow时可自动推断

ActivityRegularizer层

```
keras.layers.core.ActivityRegularization(l1=0.0, l2=0.0)
```

经过本层的数据不会有任何变化，但会基于其激活值更新损失函数值

参数

- l1：1范数正则因子（正浮点数）
- l2：2范数正则因子（正浮点数）

输入shape

任意，当使用该层作为第一层时，要指定`input_shape`

输出shape

与输入shape相同

Masking层

```
keras.layers.core.Masking(mask_value=0.0)
```

使用给定的值对输入的序列信号进行“屏蔽”，用以定位需要跳过的时间步

对于输入张量的时间步，即输入张量的第1维度（维度从0开始算，见例子），如果输入张量在该时间步上都等于`mask_value`，则该时间步将在模型接下来的所有层（只要支持masking）被跳过（屏蔽）。

如果模型接下来的一些层不支持masking，却接受到masking过的数据，则抛出异常。

例子

考虑输入数据 `x` 是一个形如(samples,timesteps,features)的张量，现将其送入LSTM层。因为你缺少时间步为3和5的信号，所以你希望将其掩盖。这时候应该：

- 赋值 `x[:,3,:] = 0.` ，`x[:,5,:] = 0.`
- 在LSTM层之前插入 `mask_value=0.` 的 `Masking` 层

```
model = Sequential()
model.add(Masking(mask_value=0., input_shape=(timesteps, features)))
model.add(LSTM(32))
```

卷积层

Conv1D层

```
keras.layers.convolutional.Conv1D(filters, kernel_size, strides=1, padding='valid', dilation_rate=1, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

一维卷积层（即时域卷积），用以在一维输入信号上进行邻域滤波。当使用该层作为首层时，需要提供关键字参数 `input_shape` 。例如 `(10,128)` 代表一个长为10的序列，序列中每个信号为128向量。而 `(None, 128)` 代表变长的128维向量序列。

该层生成将输入信号与卷积核按照单一的空域（或时域）方向进行卷积。如果 `use_bias=True` ，则还会加上一个偏置项，若 `activation` 不为None，则输出为经过激活函数的输出。

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：整数或由单个整数构成的list/tuple，卷积核的空域或时域窗长度
- strides：整数或由单个整数构成的list/tuple，为卷积的步长。任何不为1的strides均与任何不为1的dilation_rata均不兼容
- padding：补0策略，为“valid”, “same” 或“causal”，“causal”将产生因果（膨胀的）卷积，即output[t]不依赖于input[t+1：]。当对不能违反时间顺序的时序信号建模时有用。参考WaveNet: A Generative Model for Raw Audio, section 2.1.。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同。

- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：整数或由单个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rata均与任何不为1的strides均不兼容。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象

输入shape

形如（samples，steps，input_dim）的3D张量

输出shape

形如（samples，new_steps，nb_filter）的3D张量，因为有向量填充的原因， `steps` 的值会改变

【Tips】可以将Convolution1D看作Convolution2D的快捷版，对例子中（10，32）的信号进行1D卷积相当于对其进行卷积核为（filter_length, 32）的2D卷积。【@3rduncle】

Conv2D层

```
keras.layers.convolutional.Conv2D(filters, kernel_size, strides=(
1, 1), padding='valid', data_format=None, dilation_rate=(1, 1),
activation=None, use_bias=True, kernel_initializer='glorot_unifo
rm', bias_initializer='zeros', kernel_regularizer=None, bias_reg
ularizer=None, activity_regularizer=None, kernel_constraint=None
, bias_constraint=None)
```

二维卷积层，即对图像的空域卷积。该层对二维输入进行滑动窗卷积，当使用该层作为第一层时，应提供 `input_shape` 参数。例如 `input_shape = (128,128,3)` 代表128*128的彩色RGB图像（ `data_format='channels_last'` ）

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由两个整数构成的list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为1的strides均与任何不为1的dilation_rata均不兼容
- padding：补0策略，为“valid”, “same” 。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：单个整数或由两个个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rata均与任何不为1的strides均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为

例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象

输入shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的4D张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的4D张量

注意这里的输入shape指的是函数内部实现的输入shape，而非函数接口应指定的 `input_shape` ，请参考下面提供的例子。

输出shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的4D张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的4D张量

输出的行列数可能会因为填充方法而改变

SeparableConv2D层

```
keras.layers.convolutional.SeparableConv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, depth_multiplier=1, activation=None, use_bias=True, depthwise_initializer='glorot_uniform', pointwise_initializer='glorot_uniform', bias_initializer='zeros', depthwise_regularizer=None, pointwise_regularizer=None, bias_regularizer=None, activity_regularizer=None, depthwise_constraint=None, pointwise_constraint=None, bias_constraint=None)
```

该层是在深度方向上的可分离卷积。

可分离卷积首先按深度方向进行卷积（对每个输入通道分别卷积），然后逐点进行卷积，将上一步的卷积结果混合到输出通道中。参数 `depth_multiplier` 控制了在depthwise卷积（第一步）的过程中，每个输入通道信号产生多少个输出通道。

直观来说，可分离卷积可以看做讲一个卷积核分解为两个小的卷积核，或看作Inception模块的一种极端情况。

当使用该层作为第一层时，应提供 `input_shape` 参数。例如 `input_shape = (3,128,128)` 代表128*128的彩色RGB图像

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由两个个整数构成的list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为1的strides均与任何不为1的dilation_rata均不兼容
- padding：补0策略，为“valid”, “same” 。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同。

- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：单个整数或由两个个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rata均与任何不为1的strides均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。
- use_bias:布尔值，是否使用偏置项
- depth_multiplier：在按深度卷积的步骤中，每个输入通道使用多少个输出通道
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- depthwise_regularizer：施加在按深度卷积的权重上的正则项，为Regularizer对象
- pointwise_regularizer：施加在按点卷积的权重上的正则项，为Regularizer对象
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象
- depthwise_constraint：施加在按深度卷积权重上的约束项，为Constraints对象

- pointwise_constraint施加在按点卷积权重的约束项，为Constraints对象

输入shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的4D张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的4D张量

注意这里的输入shape指的是函数内部实现的输入shape，而非函数接口应指定的 `input_shape` ，请参考下面提供的例子。

输出shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的4D张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的4D张量

输出的行列数可能会因为填充方法而改变

Conv2DTranspose层

```
keras.layers.convolutional.Conv2DTranspose(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

该层是转置的卷积操作（反卷积）。需要反卷积的情况通常发生在用户想要对一个普通卷积的结果做反方向的变换。例如，将具有该卷积层输出shape的tensor转换为具有该卷积层输入shape的tensor。同时保留与卷积层兼容的连接模式。

当使用该层作为第一层时，应提供 `input_shape` 参数。例如 `input_shape = (3,128,128)` 代表128*128的彩色RGB图像

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由两个个整数构成的list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由两个整数构成的list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为1的strides均与任何不为1的dilation_rata均不兼容
- padding：补0策略，为“valid”, “same” 。“valid”代表只进行有效的卷积，即对边界数据不处理。“same”代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：单个整数或由两个个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rata均与任何不为1的strides均不兼容。
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。
- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象

- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象

输入shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的4D张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的4D张量

注意这里的输入shape指的是函数内部实现的输入shape，而非函数接口应指定的 `input_shape` ，请参考下面提供的例子。

输出shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的4D张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的4D张量

输出的行列数可能会因为填充方法而改变

参考文献

- A guide to convolution arithmetic for deep learning
- Transposed convolution arithmetic
- Deconvolutional Networks

Conv3D层

```
keras.layers.convolutional.Conv3D(filters, kernel_size, strides=(1, 1, 1), padding='valid', data_format=None, dilation_rate=(1, 1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

三维卷积对三维的输入进行滑动窗卷积，当使用该层作为第一层时，应提供 `input_shape` 参数。例如 `input_shape = (3,10,128,128)` 代表对10帧128*128的彩色RGB图像进行卷积。数据的通道位置仍然有 `data_format` 参数指定。

参数

- filters：卷积核的数目（即输出的维度）
- kernel_size：单个整数或由3个整数构成的list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。
- strides：单个整数或由3个整数构成的list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。任何不为1的strides均与任何不为1的dilation_rata均不兼容
- padding：补0策略，为"valid", "same" 。"valid"代表只进行有效的卷积，即对边界数据不处理。"same"代表保留边界处的卷积结果，通常会导致输出shape与输入shape相同。
- activation：激活函数，为预定义的激活函数名（参考激活函数），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）
- dilation_rate：单个整数或由3个个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rata均与任何不为1的strides均不兼容。
- data_format：字符串，"channels_first"或"channels_last"之一，代表数据的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，"channels_last"对应原本的"tf"，"channels_first"对应原本的"th"。以128x128x128的数据为

例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

- use_bias:布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象

输入shape

‘channels_first’模式下，输入应为形如（samples，channels，input_dim1，input_dim2, input_dim3）的5D张量

‘channels_last’模式下，输入应为形如（samples，input_dim1，input_dim2, input_dim3，channels）的5D张量

这里的输入shape指的是函数内部实现的输入shape，而非函数接口应指定的 `input_shape` 。

Cropping1D层

```
keras.layers.convolutional.Cropping1D(cropping=(1, 1))
```

在时间轴（axis1）上对1D输入（即时间序列）进行裁剪

参数

- cropping：长为2的tuple，指定在序列的首尾要裁剪掉多少个元素

输入shape

- 形如（samples，axis_to_crop，features）的3D张量

输出shape

- 形如（samples，cropped_axis，features）的3D张量

Cropping2D层

```
keras.layers.convolutional.Cropping2D(cropping=((0, 0), (0, 0)),
 data_format=None)
```

对2D输入（图像）进行裁剪，将在空域维度，即宽和高的方向上裁剪

参数

- cropping：长为2的整数tuple，分别为宽和高方向上头部与尾部需要裁剪掉的元素数
- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入shape

形如（samples，depth, first_axis_to_crop, second_axis_to_crop）

输出shape

形如(samples, depth, first_cropped_axis, second_cropped_axis)的4D张量

Cropping3D层

```
keras.layers.convolutional.Cropping3D(cropping=((1, 1), (1, 1), (1, 1)), data_format=None)
```

对2D输入（图像）进行裁剪

参数

- cropping：长为3的整数tuple，分别为三个方向上头部与尾部需要裁剪掉的元素数
- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128x128的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入shape

形如 (samples, depth, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop)的5D张量

输出shape

形如(samples, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)的5D张量

UpSampling1D层

```
keras.layers.convolutional.UpSampling1D(size=2)
```

在时间轴上，将每个时间步重复 `length` 次

参数

- size：上采样因子

输入shape

- 形如（samples，steps，features）的3D张量

输出shape

- 形如（samples，upsampled_steps，features）的3D张量

UpSampling2D层

```
keras.layers.convolutional.UpSampling2D(size=(2, 2), data_format=None)
```

将数据的行和列分别重复size[0]和size[1]次

参数

- size：整数tuple，分别为行和列上采样因子

- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的4D张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的4D张量

输出shape

‘channels_first’模式下，为形如（samples，channels, upsampled_rows, upsampled_cols）的4D张量

‘channels_last’模式下，为形如（samples，upsampled_rows, upsampled_cols，channels）的4D张量

UpSampling3D层

```
keras.layers.convolutional.UpSampling3D(size=(2, 2, 2), data_format=None)
```

将数据的三个维度上分别重复size[0]、size[1]和ize[2]次

本层目前只能在使用Theano为后端时可用

参数

- size：长为3的整数tuple，代表在三个维度上的上采样因子

- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128x128的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入shape

‘channels_first’模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的5D张量

‘channels_last’模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的5D张量

输出shape

‘channels_first’模式下，为形如（samples, channels, dim1, dim2, dim3）的5D张量

‘channels_last’模式下，为形如（samples, upsampled_dim1, upsampled_dim2, upsampled_dim3,channels,）的5D张量

ZeroPadding1D层

```
keras.layers.convolutional.ZeroPadding1D(padding=1)
```

对1D输入的首尾端（如时域序列）填充0，以控制卷积以后向量的长度

参数

- padding：整数，表示在要填充的轴的起始和结束处填充0的数目，这里要填充的轴是轴1（第1维，第0维是样本数）

输入shape

形如（samples，axis_to_pad，features）的3D张量

输出shape

形如（samples，paded_axis，features）的3D张量

ZeroPadding2D层

```
keras.layers.convolutional.ZeroPadding2D(padding=(1, 1), data_format=None)
```

对2D输入（如图片）的边界填充0，以控制卷积以后特征图的大小

参数

- padding：整数tuple，表示在要填充的轴的起始和结束处填充0的数目，这里要填充的轴是轴3和轴4（即在'th'模式下图像的行和列，在'channels_last'模式下要填充的则是轴2，3）
- data_format：字符串，"channels_first"或"channels_last"之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，"channels_last"对应原本的"tf"，"channels_first"对应原本的"th"。以128x128的RGB图像为例，"channels_first"应将数据组织为（3,128,128），而"channels_last"应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为"channels_last"。

输入shape

'channels_first'模式下，形如（samples，channels，first_axis_to_pad，second_axis_to_pad）的4D张量

'channels_last'模式下，形如（samples，first_axis_to_pad，second_axis_to_pad, channels）的4D张量

输出shape

‘channels_first’模式下，形如（samples，channels，first_paded_axis，second_paded_axis）的4D张量

‘channels_last’模式下，形如（samples，first_paded_axis，second_paded_axis, channels）的4D张量

ZeroPadding3D层

```
keras.layers.convolutional.ZeroPadding3D(padding=(1, 1, 1), data_format=None)
```

将数据的三个维度上填充0

本层目前只能在使用Theano为后端时可用

参数

padding：整数tuple，表示在要填充的轴的起始和结束处填充0的数目，这里要填充的轴是轴3，轴4和轴5，‘channels_last’模式下则是轴2，3和4

- data_format：字符串，“channels_first”或“channels_last”之一，代表数据的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128x128的数据为例，“channels_first”应将数据组织为（3,128,128,128），而“channels_last”应将数据组织为（128,128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入shape

‘channels_first’模式下，为形如（samples, channels, first_axis_to_pad，first_axis_to_pad, first_axis_to_pad,）的5D张量

‘channels_last’模式下，为形如（samples, first_axis_to_pad，first_axis_to_pad, first_axis_to_pad, channels）的5D张量

输出shape

‘channels_first’模式下，为形如（samples, channels, first_paded_axis，second_paded_axis, third_paded_axis,）的5D张量

‘channels_last’模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的5D张量

池化层

MaxPooling1D层

```
keras.layers.pooling.MaxPooling1D(pool_size=2, strides=None, pad
ding='valid')
```

对时域1D信号进行最大值池化

参数

- pool_size：整数，池化窗口大小
- strides：整数或None，下采样因子，例如设2将会使得输出shape为输入的一半，若为None则默认值为pool_size。
- padding：'valid'或者'same'

输入shape

- 形如（samples，steps，features）的3D张量

输出shape

- 形如（samples，downsampled_steps，features）的3D张量

MaxPooling2D层

```
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=None
, padding='valid', data_format=None)
```

为空域信号施加最大值池化

参数

- pool_size：整数或长为2的整数tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半。为整数意为各个维度值相同且为该数字。
- strides：整数或长为2的整数tuple，或者None，步长值。
- border_mode：'valid'或者'same'
- data_format：字符串，"channels_first"或"channels_last"之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，"channels_last"对应原本的"tf"，"channels_first"对应原本的"th"。以128x128的RGB图像为例，"channels_first"应将数据组织为（3,128,128），而"channels_last"应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为"channels_last"。

输入shape

'channels_first'模式下，为形如（samples，channels, rows，cols）的4D张量

'channels_last'模式下，为形如（samples，rows, cols，channels）的4D张量

输出shape

'channels_first'模式下，为形如（samples，channels, pooled_rows, pooled_cols）的4D张量

'channels_last'模式下，为形如（samples，pooled_rows, pooled_cols，channels）的4D张量

MaxPooling3D层

```
keras.layers.pooling.MaxPooling3D(pool_size=(2, 2, 2), strides=None, padding='valid', data_format=None)
```

为3D信号（空域或时空域）施加最大值池化

本层目前只能在使用Theano为后端时可用

参数

- pool_size：整数或长为3的整数tuple，代表在三个维度上的下采样因子，如取（2，2，2）将使信号在每个维度都变为原来的一半长。
- strides：整数或长为3的整数tuple，或者None，步长值。
- padding：'valid'或者'same'
- data_format：字符串，"channels_first"或"channels_last"之一，代表数据的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，"channels_last"对应原本的"tf"，"channels_first"对应原本的"th"。以128x128x128的数据为例，"channels_first"应将数据组织为（3,128,128,128），而"channels_last"应将数据组织为（128,128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为"channels_last"。

输入shape

'channels_first'模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的5D张量

'channels_last'模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的5D张量

输出shape

'channels_first'模式下，为形如（samples, channels, pooled_dim1, pooled_dim2, pooled_dim3）的5D张量

'channels_last'模式下，为形如（samples, pooled_dim1, pooled_dim2, pooled_dim3,channels,）的5D张量

AveragePooling1D层

```
keras.layers.pooling.AveragePooling1D(pool_size=2, strides=None,
 padding='valid')
```

对时域1D信号进行平均值池化

参数

- pool_size：整数，池化窗口大小
- strides：整数或None，下采样因子，例如设2将会使得输出shape为输入的一半，若为None则默认值为pool_size。
- padding：‘valid’或者‘same’

输入shape

- 形如（samples，steps，features）的3D张量

输出shape

- 形如（samples，downsampled_steps，features）的3D张量

AveragePooling2D层

```
keras.layers.pooling.AveragePooling2D(pool_size=(2, 2), strides=
None, padding='valid', data_format=None)
```

为空域信号施加平均值池化

参数

- pool_size：整数或长为2的整数tuple，代表在两个方向（竖直，水平）上的下采样因子，如取（2，2）将使图片在两个维度上均变为原长的一半。为整数意为各个维度值相同且为该数字。
- strides：整数或长为2的整数tuple，或者None，步长值。
- border_mode：'valid'或者'same'
- data_format：字符串，"channels_first"或"channels_last"之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，"channels_last"对应原本的"tf"，"channels_first"对应原本的"th"。以128x128的RGB图像为例，"channels_first"应将数据组织为（3,128,128），而"channels_last"应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为"channels_last"。

输入shape

'channels_first'模式下，为形如（samples，channels, rows，cols）的4D张量

'channels_last'模式下，为形如（samples，rows, cols，channels）的4D张量

输出shape

'channels_first'模式下，为形如（samples，channels, pooled_rows, pooled_cols）的4D张量

'channels_last'模式下，为形如（samples，pooled_rows, pooled_cols，channels）的4D张量

AveragePooling3D层

```
keras.layers.pooling.AveragePooling3D(pool_size=(2, 2, 2), strides=None, padding='valid', data_format=None)
```

为3D信号（空域或时空域）施加平均值池化

本层目前只能在使用Theano为后端时可用

参数

- pool_size：整数或长为3的整数tuple，代表在三个维度上的下采样因子，如取（2，2，2）将使信号在每个维度都变为原来的一半长。
- strides：整数或长为3的整数tuple，或者None，步长值。
- padding：'valid'或者'same'
- data_format：字符串，"channels_first"或"channels_last"之一，代表数据的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，"channels_last"对应原本的"tf"，"channels_first"对应原本的"th"。以128x128x128的数据为例，"channels_first"应将数据组织为（3,128,128,128），而"channels_last"应将数据组织为（128,128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为"channels_last"。

'channels_first'模式下，为形如（samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3）的5D张量

'channels_last'模式下，为形如（samples, len_pool_dim1, len_pool_dim2, len_pool_dim3，channels, ）的5D张量

输出shape

'channels_first'模式下，为形如（samples, channels, pooled_dim1, pooled_dim2, pooled_dim3）的5D张量

'channels_last'模式下，为形如（samples, pooled_dim1, pooled_dim2, pooled_dim3,channels,）的5D张量

GlobalMaxPooling1D层

```
keras.layers.pooling.GlobalMaxPooling1D()
```

对于时间信号的全局最大池化

输入shape

- 形如（samples，steps，features）的3D张量

输出shape

- 形如(samples, features)的2D张量

GlobalAveragePooling1D层

```
keras.layers.pooling.GlobalAveragePooling1D()
```

为时域信号施加全局平均值池化

输入shape

- 形如（samples，steps，features）的3D张量

输出shape

- 形如(samples, features)的2D张量

GlobalMaxPooling2D层

```
keras.layers.pooling.GlobalMaxPooling2D(dim_ordering='default')
```

为空域信号施加全局最大值池化

参数

- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的4D张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的4D张量

输出shape

形如(nb_samples, channels)的2D张量

GlobalAveragePooling2D层

```
keras.layers.pooling.GlobalAveragePooling2D(dim_ordering='default')
```

为空域信号施加全局平均值池化

参数

- data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channels_last”。

输入shape

‘channels_first’模式下，为形如（samples，channels, rows，cols）的4D张量

‘channels_last’模式下，为形如（samples，rows, cols，channels）的4D张量

输出shape

形如(nb_samples, channels)的2D张量

局部连接层LocallyConnceted

LocallyConnected1D层

```
keras.layers.local.LocallyConnected1D(filters, kernel_size, strides=1, padding='valid', data_format=None, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

```
### 参数

* filters：卷积核的数目（即输出的维度）

* kernel_size：整数或由单个整数构成的list/tuple，卷积核的空域或时域窗长度

* strides：整数或由单个整数构成的list/tuple，为卷积的步长。任何不为1的strides均与任何不为1的dilation_rata均不兼容

* padding：补0策略，目前仅支持`valid`（大小写敏感），`same`可能会在将来支持。

* activation：激活函数，为预定义的激活函数名（参考[激活函数](../other/activations)），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

* dilation_rate：整数或由单个整数构成的list/tuple，指定dilated convolution中的膨胀比例。任何不为1的dilation_rata均与任何不为1的strides均不兼容。

* use_bias:布尔值，是否使用偏置项

* kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用
```

````
于初始化权重的初始化器。参考[initializers](../other/initializations)

* bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考[initializers](../other/initializations)

* kernel_regularizer：施加在权重上的正则项，为[Regularizer](../other/regularizers)对象

* bias_regularizer：施加在偏置向量上的正则项，为[Regularizer](../other/regularizers)对象

* activity_regularizer：施加在输出上的正则项，为[Regularizer](../other/regularizers)对象

* kernel_constraints：施加在权重上的约束项，为[Constraints](../other/constraints)对象

* bias_constraints：施加在偏置上的约束项，为[Constraints](../other/constraints)对象

### 输入shape

形如（samples，steps，input_dim）的3D张量

### 输出shape

形如（samples，new_steps，nb_filter）的3D张量，因为有向量填充的原因，```steps```的值会改变



***

## LocallyConnected2D层
```python
keras.layers.local.LocallyConnected2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
````
````

````
### 参数

* filters：卷积核的数目（即输出的维度）

* kernel_size：单个整数或由两个整数构成的list/tuple，卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。

* strides：单个整数或由两个整数构成的list/tuple，为卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。

* padding：补0策略，目前仅支持`valid`（大小写敏感），`same`可能会在将来支持。

* activation：激活函数，为预定义的激活函数名（参考[激活函数](../other/activations)），或逐元素（element-wise）的Theano函数。如果不指定该参数，将不会使用任何激活函数（即使用线性激活函数：a(x)=x）

* data_format：字符串，“channels_first”或“channels_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channels_last”对应原本的“tf”，“channels_first”对应原本的“th”。以128x128的RGB图像为例，“channels_first”应将数据组织为（3,128,128），而“channels_last”应将数据组织为（128,128,3）。该参数的默认值是```~/.keras/keras.json```中设置的值，若从未设置过，则为“channels_last”。

* use_bias:布尔值，是否使用偏置项

* kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考[initializers](../other/initializations)

* bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考[initializers](../other/initializations)

* kernel_regularizer：施加在权重上的正则项，为[Regularizer](../other/regularizers)对象

* bias_regularizer：施加在偏置向量上的正则项，为[Regularizer](../other/regularizers)对象
````

````
* activity_regularizer：施加在输出上的正则项，为[Regularizer](../other/regularizers)对象

* kernel_constraints：施加在权重上的约束项，为[Constraints](../other/constraints)对象

* bias_constraints：施加在偏置上的约束项，为[Constraints](../other/constraints)对象

### 输入shape

‘channels_first’模式下，输入形如（samples,channels，rows，cols）的4D张量

‘channels_last’模式下，输入形如（samples，rows，cols，channels）的4D张量

注意这里的输入shape指的是函数内部实现的输入shape，而非函数接口应指定的```input_shape```，请参考下面提供的例子。

### 输出shape

‘channels_first’模式下，为形如（samples，nb_filter, new_rows, new_cols）的4D张量

‘channels_last’模式下，为形如（samples，new_rows, new_cols，nb_filter）的4D张量

输出的行列数可能会因为填充方法而改变

### 例子

```python
# apply a 3x3 unshared weights convolution with 64 output filters on a 32x32 image
# with `data_format="channels_last"`:
model = Sequential()
model.add(LocallyConnected2D(64, (3, 3), input_shape=(32, 32, 3)))
# now model.output_shape == (None, 30, 30, 64)
````
````

```
# notice that this layer will consume (30*30)*(3*3*3*64) + (30*3
0)*64 parameters

# add a 3x3 unshared weights convolution on top, with 32 output
filters:
model.add(LocallyConnected2D(32, (3, 3)))
# now model.output_shape == (None, 28, 28, 32)
```

循环层Recurrent

Recurrent层

```
keras.layers.recurrent.Recurrent(return_sequences=False, go_backwards=False, stateful=False, unroll=False, implementation=0)
```

这是循环层的抽象类，请不要在模型中直接应用该层（因为它是抽象类，无法实例化任何对象）。请使用它的子类 `LSTM` ， `GRU` 或 `SimpleRNN` 。

所有的循环层（ `LSTM` , `GRU` , `SimpleRNN` ）都服从本层的性质，并接受本层指定的所有关键字参数。

参数

- weights：numpy array的list，用以初始化权重。该list形如 `[(input_dim, output_dim),(output_dim, output_dim),(output_dim,)]`
- return_sequences：布尔值，默认 `False` ，控制返回类型。若为 `True` 则返回整个序列，否则仅返回输出序列的最后一个输出
- go_backwards：布尔值，默认为 `False` ，若为 `True` ，则逆向处理输入序列并返回逆序后的序列
- stateful：布尔值，默认为 `False` ，若为 `True` ，则一个batch中下标为i的样本的最终状态将会用作下一个batch同样下标的样本的初始状态。
- unroll：布尔值，默认为 `False` ，若为 `True` ，则循环层将被展开，否则就使用符号化的循环。当使用TensorFlow为后端时，循环网络本来就是展开的，因此该层不做任何事情。层展开会占用更多的内存，但会加速RNN的运算。层展开只适用于短序列。
- implementation：0，1或2， 若为0，则RNN将以更少但是更大的矩阵乘法实现，因此在CPU上运行更快，但消耗更多的内存。如果设为1，则RNN将以更多但更小的矩阵乘法实现，因此在CPU上运行更慢，在GPU上运行更快，并且

消耗更少的内存。如果设为2（仅LSTM和GRU可以设为2），则RNN将把输入门、遗忘门和输出门合并为单个矩阵，以获得更加在GPU上更加高效的实现。注意，RNN dropout必须在所有门上共享，并导致正则效果性能微弱降低。

- input_dim：输入维度，当使用该层为模型首层时，应指定该值（或等价的指定input_shape)
- input_length：当输入序列的长度固定时，该参数为输入序列的长度。当需要在该层后连接 `Flatten` 层，然后又要连接 `Dense` 层时，需要指定该参数，否则全连接的输出无法计算出来。注意，如果循环层不是网络的第一层，你需要在网络的第一层中指定序列的长度（通过 `input_shape` 指定）。

输入shape

形如（samples，timesteps，input_dim）的3D张量

输出shape

如果 `return_sequences=True` ：返回形如（samples，timesteps，output_dim）的3D张量

否则，返回形如（samples，output_dim）的2D张量

例子

```
# as the first layer in a Sequential model
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))
# now model.output_shape == (None, 32)
# note: `None` is the batch dimension.

# the following is identical:
model = Sequential()
model.add(LSTM(32, input_dim=64, input_length=10))

# for subsequent layers, no need to specify the input size:
         model.add(LSTM(16))

# to stack recurrent layers, you must use return_sequences=True
# on any recurrent layer that feeds into another recurrent layer.

# note that you only need to specify the input size on the first
 layer.
model = Sequential()
model.add(LSTM(64, input_dim=64, input_length=10, return_sequenc
es=True))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(10))
```

指定RNN初始状态的注意事项

可以通过设置 `initial_state` 用符号式的方式指定RNN层的初始状态。即， `initial_stat` 的值应该为一个tensor或一个tensor列表，代表RNN层的初始状态。

也可以通过设置 `reset_states` 参数用数值的方法设置RNN的初始状态，状态的值应该为numpy数组或numpy数组的列表，代表RNN层的初始状态。

屏蔽输入数据（Masking）

循环层支持通过时间步变量对输入数据进行Masking，如果想将输入数据的一部分屏蔽掉，请使用Embedding层并将参数 `mask_zero` 设为 `True` 。

使用状态RNN的注意事项

可以将RNN设置为'stateful'，意味着由每个batch计算出的状态都会被重用于初始化下一个batch的初始状态。状态RNN假设连续的两个batch之中，相同下标的元素有一一映射关系。

要启用状态RNN，请在实例化层对象时指定参数 `stateful=True` ，并在Sequential模型使用固定大小的batch：通过在模型的第一层传入 `batch_size=(...)` 和 `input_shape` 来实现。在函数式模型中，对所有的输入都要指定相同的 `batch_size` 。

如果要将循环层的状态重置，请调用 `.reset_states()` ，对模型调用将重置模型中所有状态RNN的状态。对单个层调用则只重置该层的状态。

SimpleRNN层

```
keras.layers.recurrent.SimpleRNN(units, activation='tanh', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)
```

全连接RNN网络，RNN的输出会被回馈到输入

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers

- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- recurrent_regularizer：施加在循环核上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- recurrent_constraints：施加在循环核上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象
- dropout：0~1之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1之间的浮点数，控制循环状态的线性变换的神经元断开比例

参考文献

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

---

GRU层

```
keras.layers.recurrent.GRU(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)
```

门限循环单元（详见参考文献）

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象
- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- recurrent_regularizer：施加在循环核上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- recurrent_constraints：施加在循环核上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象
- dropout：0~1之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1之间的浮点数，控制循环状态的线性变换的神经元断开比例

参考文献

- On the Properties of Neural Machine Translation: Encoder–Decoder Approaches

- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

LSTM层

```
keras.layers.recurrent.LSTM(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0)
```

Keras长短期记忆模型，关于此算法的详情，请参考本教程

参数

- units：输出维度
- activation：激活函数，为预定义的激活函数名（参考激活函数）
- recurrent_activation: 为循环步施加的激活函数（参考激活函数）
- use_bias: 布尔值，是否使用偏置项
- kernel_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- recurrent_initializer：循环核的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- bias_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- kernel_regularizer：施加在权重上的正则项，为Regularizer对象

- bias_regularizer：施加在偏置向量上的正则项，为Regularizer对象
- recurrent_regularizer：施加在循环核上的正则项，为Regularizer对象
- activity_regularizer：施加在输出上的正则项，为Regularizer对象
- kernel_constraints：施加在权重上的约束项，为Constraints对象
- recurrent_constraints：施加在循环核上的约束项，为Constraints对象
- bias_constraints：施加在偏置上的约束项，为Constraints对象
- dropout：0~1之间的浮点数，控制输入线性变换的神经元断开比例
- recurrent_dropout：0~1之间的浮点数，控制循环状态的线性变换的神经元断开比例

参考文献

- Long short-term memory（original 1997 paper）
- Learning to forget: Continual prediction with LSTM
- Supervised sequence labelling with recurrent neural networks
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

嵌入层 Embedding

Embedding层

```
keras.layers.embeddings.Embedding(input_dim, output_dim, embeddings_initializer='uniform', embeddings_regularizer=None, activity_regularizer=None, embeddings_constraint=None, mask_zero=False, input_length=None)
```

嵌入层将正整数（下标）转换为具有固定大小的向量，如[[4],[20]]->[[0.25,0.1],[0.6,-0.2]]

Embedding层只能作为模型的第一层

参数

- input_dim：大或等于0的整数，字典长度，即输入数据最大下标+1
- output_dim：大于0的整数，代表全连接嵌入的维度
- embeddings_initializer: 嵌入矩阵的初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。参考initializers
- embeddings_regularizer: 嵌入矩阵的正则项，为Regularizer对象
- embeddings_constraint: 嵌入矩阵的约束项，为Constraints对象
- mask_zero：布尔值，确定是否将输入中的'0'看作是应该被忽略的'填充'（padding）值，该参数在使用递归层处理变长输入时有用。设置为 `True` 的话，模型中后续的层必须都支持masking，否则会抛出异常。如果该值为True，则下标0在字典中不可用，input_dim应设置为|vocabulary| + 2。
- input_length：当输入序列的长度固定时，该值为其长度。如果要在该层后接 `Flatten` 层，然后接 `Dense` 层，则必须指定该参数，否则 `Dense` 层的输出维度无法自动推断。

输入shape

形如（samples，sequence_length）的2D张量

输出shape

形如(samples, sequence_length, output_dim)的3D张量

例子

```
model = Sequential()
model.add(Embedding(1000, 64, input_length=10))
# the model will take as input an integer matrix of size (batch,
 input_length).
# the largest integer (i.e. word index) in the input should be n
o larger than 999 (vocabulary size).
# now model.output_shape == (None, 10, 64), where None is the ba
tch dimension.

input_array = np.random.randint(1000, size=(32, 10))

model.compile('rmsprop', 'mse')
output_array = model.predict(input_array)
assert output_array.shape == (32, 10, 64)
```

参考文献

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

Merge层

Merge层提供了一系列用于融合两个层或两个张量的层对象和方法。以大写首字母开头的是Layer类，以小写字母开头的是张量的函数。小写字母开头的张量函数在内部实际上是调用了大写字母开头的层。

Add

```
keras.layers.merge.Add()
```

将Layer that adds a list of inputs.

该层接收一个列表的同shape张量，并返回它们的和，shape不变。

Multiply

``python keras.layers.merge.Multiply()

````
该层接收一个列表的同shape张量，并返回它们的逐元素积的张量，shape不变。

## Average
```python
keras.layers.merge.Average()
````

该层接收一个列表的同shape张量，并返回它们的逐元素均值，shape不变。

Maximum

```
keras.layers.merge.Maximum()
```

该层接收一个列表的同shape张量，并返回它们的逐元素最大值，shape不变。
```

Concatenate

```
keras.layers.merge.Concatenate(axis=-1)
```

该层接收一个列表的同shape张量，并返回它们的按照给定轴相接构成的向量。

参数

- axis: 想接的轴
- **kwargs: 普通的Layer关键字参数

Dot

```
keras.layers.merge.Dot(axes, normalize=False)
```

计算两个tensor中样本的张量乘积。例如，如果两个张量 `a` 和 `b` 的shape都为（batch_size, n），则输出为形如（batch_size,1）的张量，结果张量每个batch的数据都是a[i,:]和b[i,:]的矩阵（向量）点积。

参数

- axes: 整数或整数的tuple，执行乘法的轴。
- normalize: 布尔值，是否沿执行成绩的轴做L2规范化，如果设为True，那么乘积的输出是两个样本的余弦相似性。
- **kwargs: 普通的Layer关键字参数

add

```
add(inputs)
```

Add层的函数式包装

参数：

- inputs: 长度至少为2的张量列表A
- **kwargs: 普通的Layer关键字参数

返回值

输入列表张量之和

multiply

``python multiply(inputs)

```
Multiply的函数包装

###参数：

* inputs: 长度至少为2的张量列表
* **kwargs: 普通的Layer关键字参数
###返回值

输入列表张量之逐元素积

## average
```python
average(inputs)
```

Average的函数包装

参数：

- inputs: 长度至少为2的张量列表
- **kwargs: 普通的Layer关键字参数

返回值
```

输入列表张量之逐元素均值

maximum

```
maximum(inputs)
```

Maximum的函数包装

参数：

- inputs: 长度至少为2的张量列表
- **kwargs: 普通的Layer关键字参数

返回值

输入列表张量之逐元素均值

concatenate

```
concatenate(inputs, axis=-1))
```

Concatenate的函数包装

参数

- inputs: 长度至少为2的张量列
- axis: 相接的轴
- **kwargs: 普通的Layer关键字参数

dot

```
dot(inputs, axes, normalize=False)
```

Dot的函数包装

参数

- inputs: 长度至少为2的张量列
- axes: 整数或整数的tuple，执行乘法的轴。
- normalize: 布尔值，是否沿执行成绩的轴做L2规范化，如果设为True，那么乘积的输出是两个样本的余弦相似性。
- **kwargs: 普通的Layer关键字参数

高级激活层Advanced Activation

LeakyReLU层

```
keras.layers.advanced_activations.LeakyReLU(alpha=0.3)
```

LeakyRelU是修正线性单元（Rectified Linear Unit，ReLU）的特殊版本，当不激活时，LeakyReLU仍然会有非零输出值，从而获得一个小梯度，避免ReLU可能出现的神经元“死亡”现象。即，`f(x)=alpha * x for x < 0`，`f(x) = x for x>=0`

参数

- alpha：大于0的浮点数，代表激活函数图像中第三象限线段的斜率

输入shape

任意，当使用该层为模型首层时需指定`input_shape`参数

输出shape

与输入相同

参考文献

Rectifier Nonlinearities Improve Neural Network Acoustic Models

---

PReLU层

```
keras.layers.advanced_activations.PReLU(alpha_initializer='zeros'
, alpha_regularizer=None, alpha_constraint=None, shared_axes=None
)
```

该层为参数化的ReLU（Parametric ReLU），表达式是：`f(x) = alpha * x for x < 0`，`f(x) = x for x>=0`，此处的`alpha`为一个与xshape相同的可学习的参数向量。

参数

- alpha_initializer：alpha的初始化函数
- alpha_regularizer：alpha的正则项
- alpha_constraint：alpha的约束项
- shared_axes：该参数指定的轴将共享同一组科学系参数，例如假如输入特征图是从2D卷积过来的，具有形如`(batch, height, width, channels)`这样的shape，则或许你会希望在空域共享参数，这样每个filter就只有一组参数，设定`shared_axes=[1,2]`可完成该目标

输入shape

任意，当使用该层为模型首层时需指定`input_shape`参数

输出shape

与输入相同

参考文献

- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

---

ELU层

```
keras.layers.advanced_activations.ELU(alpha=1.0)
```

ELU层是指数线性单元（Exponential Linera Unit），表达式为： 该层为参数化的ReLU（Parametric ReLU），表达式是：`f(x) = alpha * (exp(x) - 1.) for x < 0`，`f(x) = x for x>=0`

参数

- alpha：控制负因子的参数

输入shape

任意，当使用该层为模型首层时需指定`input_shape`参数

输出shape

与输入相同

参考文献

- >Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)

ThresholdedReLU层

```
keras.layers.advanced_activations.ThresholdedReLU(theta=1.0)
```

该层是带有门限的ReLU，表达式是：`f(x) = x for x > theta`，`f(x) = 0 otherwise`

参数

- theata：大或等于0的浮点数，激活门限位置

输入shape

任意，当使用该层为模型首层时需指定`input_shape`参数

输出shape

与输入相同

参考文献

- Zero-Bias Autoencoders and the Benefits of Co-Adapting Features

（批）规范化BatchNormalization

BatchNormalization层

```
keras.layers.normalization.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True, beta_initializer='zeros', gamma_initializer='ones', moving_mean_initializer='zeros', moving_variance_initializer='ones', beta_regularizer=None, gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)
```

该层在每个batch上将前一层的激活值重新规范化，即使得其输出数据的均值接近0，其标准差接近1

参数

- axis: 整数，指定要规范化的轴，通常为特征轴。例如在进行 `data_format="channels_first` 的2D卷积后，一般会设axis=1。
- momentum: 动态均值的动量
- epsilon：大于0的小浮点数，用于防止除0错误
- center: 若设为True，将会将beta作为偏置加上去，否则忽略参数beta
- scale: 若设为True，则会乘以gamma，否则不使用gamma。当下一层是线性的时，可以设False，因为scaling的操作将被下一层执行。
- beta_initializer：beta权重的初始方法
- gamma_initializer: gamma的初始化方法
- moving_mean_initializer: 动态均值的初始化方法
- moving_variance_initializer: 动态方差的初始化方法
- beta_regularizer: 可选的beta正则
- gamma_regularizer: 可选的gamma正则
- beta_constraint: 可选的beta约束
- gamma_constraint: 可选的gamma约束

输入shape

任意，当使用本层为模型首层时，指定`input_shape`参数时有意义。

输出shape

与输入shape相同

参考文献

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

【Tips】BN层的作用

（1）加速收敛（2）控制过拟合，可以少用或不用Dropout和正则（3）降低网络对初始化权重不敏感（4）允许使用较大的学习率

噪声层Noise

GaussianNoise层

```
keras.layers.noise.GaussianNoise(stddev)
```

为数据施加0均值，标准差为 `stddev` 的加性高斯噪声。该层在克服过拟合时比较有用，你可以将它看作是随机的数据提升。高斯噪声是需要对输入数据进行破坏时的自然选择。

因为这是一个起正则化作用的层，该层只在训练时才有效。

参数

- stddev：浮点数，代表要产生的高斯噪声标准差

输入shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出shape

与输入相同

GaussianDropout层

```
keras.layers.noise.GaussianDropout(rate)
```

为层的输入施加以1为均值，标准差为 `sqrt(rate/(1-rate)` 的乘性高斯噪声

因为这是一个起正则化作用的层，该层只在训练时才有效。

参数

- rate：浮点数，断连概率，与Dropout层相同

输入shape

任意，当使用该层为模型首层时需指定 `input_shape` 参数

输出shape

与输入相同

参考文献

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting

包装器Wrapper

TimeDistributed包装器

```
keras.layers.wrappers.TimeDistributed(layer)
```

该包装器可以把一个层应用到输入的每一个时间步上

参数

- layer：Keras层对象

输入至少为3D张量，下标为1的维度将被认为是时间维

例如，考虑一个含有32个样本的batch，每个样本都是10个向量组成的序列，每个向量长为16，则其输入维度为 `(32,10,16)` ，其不包含batch大小的 `input_shape` 为 `(10,16)`

我们可以使用包装器 `TimeDistributed` 包装 `Dense` ，以产生针对各个时间步信号的独立全连接：

```
# as the first layer in a model
model = Sequential()
model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
# now model.output_shape == (None, 10, 8)

# subsequent layers: no need for input_shape
model.add(TimeDistributed(Dense(32)))
# now model.output_shape == (None, 10, 32)
```

程序的输出数据shape为 `(32,10,8)`

使用 `TimeDistributed` 包装 `Dense` 严格等价于 `layers.TimeDistribuedDense` 。不同的是包装器 `TimeDistribued` 还可以对别的层进行包装，如这里对 `Convolution2D` 包装：

```
model = Sequential()
model.add(TimeDistributed(Convolution2D(64, 3, 3), input_shape=(
10, 3, 299, 299)))
```

Bidirectional包装器

```
keras.layers.wrappers.Bidirectional(layer, merge_mode='concat',
weights=None)
```

双向RNN包装器

参数

- layer： `Recurrent` 对象
- merge_mode：前向和后向RNN输出的结合方式，为 `sum` , `mul` , `concat` , `ave` 和 `None` 之一，若设为None，则返回值不结合，而是以列表的形式返回

例子

```
model = Sequential()
model.add(Bidirectional(LSTM(10, return_sequences=True), input_s
hape=(5, 10)))
model.add(Bidirectional(LSTM(10)))
model.add(Dense(5))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmspro
p')
```

编写自己的层

对于简单的定制操作，我们或许可以通过使用 `layers.core.Lambda` 层来完成。但对于任何具有可训练权重的定制层，你应该自己来实现。

这里是一个Keras2的层应该具有的框架结构(如果你的版本更旧请升级)，要定制自己的层，你需要实现下面三个方法

- `build(input_shape)` ：这是定义权重的方法，可训练的权应该在这里被加入列表 `` `self.trainable_weights `` 中。其他的属性还包括 `self.non_trainabe_weights` （列表）和 `self.updates` （需要更新的形如（tensor, new_tensor）的tuple的列表）。你可以参考 `BatchNormalization` 层的实现来学习如何使用上面两个属性。这个方法必须设置 `self.built = True` ，可通过调用 `super([layer],self).build()` 实现
- `call(x)` ：这是定义层功能的方法，除非你希望你写的层支持masking，否则你只需要关心 `call` 的第一个参数：输入张量
- `compute_output_shape(input_shape)` ：如果你的层修改了输入数据的shape，你应该在这里指定shape变化的方法，这个函数使得Keras可以做自动shape推断

```
from keras import backend as K
from keras.engine.topology import Layer
import numpy as np

class MyLayer(Layer):

    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Create a trainable weight variable for this layer.
        self.kernel = self.add_weight(shape=(input_shape[1], self.output_dim),
                                      initializer='uniform',
                                      trainable=True)
        super(MyLayer, self).build(input_shape)  # Be sure to call this somewhere!

    def call(self, x):
        return K.dot(x, self.kernel)

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)
```

现存的Keras层代码可以为你的实现提供良好参考，阅读源代码吧！

数据预处理

序列预处理

填充序列pad_sequences

```
keras.preprocessing.sequence.pad_sequences(sequences, maxlen=None, dtype='int32',
    padding='pre', truncating='pre', value=0.)
```

将长为 `nb_samples` 的序列（标量序列）转化为形如 `(nb_samples,nb_timesteps)` 2D numpy array。如果提供了参数 `maxlen` ，`nb_timesteps=maxlen` ，否则其值为最长序列的长度。其他短于该长度的序列都会在后部填充0以达到该长度。长于 `nb_timesteps` 的序列将会被截断，以使其匹配目标长度。padding和截断发生的位置分别取决于 `padding` 和 `truncating` .

参数

- sequences：浮点数或整数构成的两层嵌套列表
- maxlen：None或整数，为序列的最大长度。大于此长度的序列将被截短，小于此长度的序列将在后部填0.
- dtype：返回的numpy array的数据类型
- padding：'pre'或'post'，确定当需要补0时，在序列的起始还是结尾补
- truncating：'pre'或'post'，确定当需要截断序列时，从起始还是结尾截断
- value：浮点数，此值将在填充时代替默认的填充值0

返回值

返回形如 `(nb_samples,nb_timesteps)` 的2D张量

跳字skipgrams

```
keras.preprocessing.sequence.skipgrams(sequence, vocabulary_size
,
    window_size=4, negative_samples=1., shuffle=True,
    categorical=False, sampling_table=None)
```

skipgrams将一个词向量下标的序列转化为下面的一对tuple：

- 对于正样本，转化为（word，word in the same window）
- 对于负样本，转化为（word，random word from the vocabulary）

【Tips】根据维基百科，n-gram代表在给定序列中产生连续的n项，当序列句子时，每项就是单词，此时n-gram也称为shingles。而skip-gram的推广，skip-gram产生的n项子序列中，各个项在原序列中不连续，而是跳了k个字。例如，对于句子：

“the rain in Spain falls mainly on the plain”

其 2-grams为子序列集合：

the rain，rain in，in Spain，Spain falls，falls mainly，mainly on，on the，the plain

其 1-skip-2-grams为子序列集合：

the in, rain Spain, in falls, Spain mainly, falls on, mainly the, on plain.

更多详情请参考Efficient Estimation of Word Representations in Vector Space

参数

- sequence：下标的列表，如果使用sampling_tabel，则某个词的下标应该为它在数据库中的顺序。（从1开始）
- vocabulary_size：整数，字典大小
- window_size：整数，正样本对之间的最大距离

- negative_samples：大于0的浮点数，等于0代表没有负样本，等于1代表负样本与正样本数目相同，以此类推（即负样本的数目是正样本的 `negative_samples` 倍）
- shuffle：布尔值，确定是否随机打乱样本
- categorical：布尔值，确定是否要使得返回的标签具有确定类别
- sampling_table：形如 `(vocabulary_size,)` 的numpy array，其中 `sampling_table[i]` 代表没有负样本或随机负样本。等于1为与正样本的数目相同 采样到该下标为i的单词的概率（假定该单词是数据库中第i常见的单词）

输出

函数的输出是一个 `(couples,labels)` 的元组，其中：

- `couples` 是一个长为2的整数列表：`[word_index,other_word_index]`
- `labels` 是一个仅由0和1构成的列表，1代表 `other_word_index` 在 `word_index` 的窗口，0代表 `other_word_index` 是词典里的随机单词。
- 如果设置 `categorical` 为 `True` ，则标签将以one-hot的方式给出，即1变为[0,1]，0变为[1,0]

获取采样表**make_sampling_table**

```
keras.preprocessing.sequence.make_sampling_table(size, sampling_
factor=1e-5)
```

该函数用以产生 `skipgrams` 中所需要的参数 `sampling_table` 。这是一个长为 `size` 的向量，`sampling_table[i]` 代表采样到数据集中第i常见的词的概率（为平衡期起见，对于越经常出现的词，要以越低的概率采到它）

参数

- size：词典的大小
- sampling_factor：此值越低，则代表采样时更缓慢的概率衰减（即常用的词会被以更低的概率被采到），如果设置为1，则代表不进行下采样，即所有样本被采样到的概率都是1。

文本预处理

句子分割text_to_word_sequence

```
keras.preprocessing.text.text_to_word_sequence(text,
    filters=base_filter(), lower=True, split=" ")
```

本函数将一个句子拆分成单词构成的列表

参数

- text：字符串，待处理的文本
- filters：需要滤除的字符的列表或连接形成的字符串，例如标点符号。默认值为 `base_filter()` ，包含标点符号，制表符和换行符等
- lower：布尔值，是否将序列设为小写形式
- split：字符串，单词的分隔符，如空格

返回值

字符串列表

---

one-hot编码

```
keras.preprocessing.text.one_hot(text, n,
    filters=base_filter(), lower=True, split=" ")
```

本函数将一段文本编码为one-hot形式的码，即仅记录词在词典中的下标。

【Tips】 从定义上，当字典长为n时，每个单词应形成一个长为n的向量，其中仅有单词本身在字典中下标的位置为1，其余均为0，这称为one-hot。

为了方便起见，函数在这里仅把“1”的位置，即字典中词的下标记录下来。

参数

- n：整数，字典长度

返回值

整数列表，每个整数是[1,n]之间的值，代表一个单词（不保证唯一性，即如果词典长度不够，不同的单词可能会被编为同一个码）。

分词器Tokenizer

```
keras.preprocessing.text.Tokenizer(num_words=None, filters=base_
filter(),
    lower=True, split=" ")
```

Tokenizer是一个用于向量化文本，或将文本转换为序列（即单词在字典中的下标构成的列表，从1算起）的类。

构造参数

- 与 `text_to_word_sequence` 同名参数含义相同
- nb_words：None或整数，处理的最大单词数量。若被设置为整数，则分词器将被限制为处理数据集中最常见的 `nb_words` 个单词

类方法

- fit_on_texts(texts)
    - texts：要用以训练的文本列表
- texts_to_sequences(texts)
    - texts：待转为序列的文本列表

    - 返回值：序列的列表，列表中每个序列对应于一段输入文本

- texts_to_sequences_generator(texts)
    - 本函数是 `texts_to_sequences` 的生成器函数版
    - texts：待转为序列的文本列表
    - 返回值：每次调用返回对应于一段输入文本的序列
- texts_to_matrix(texts, mode)：
    - texts：待向量化的文本列表
    - mode：'binary'，'count'，'tfidf'，'freq'之一，默认为'binary'
    - 返回值：形如 `(len(texts), nb_words)` 的numpy array
- fit_on_sequences(sequences):
    - sequences：要用以训练的序列列表
- sequences_to_matrix(sequences):
    - sequences：待向量化的序列列表
    - mode：'binary'，'count'，'tfidf'，'freq'之一，默认为'binary'
    - 返回值：形如 `(len(sequences), nb_words)` 的numpy array

属性

- word_counts:字典，将单词（字符串）映射为它们在训练期间出现的次数。仅在调用fit_on_texts之后设置。
- word_docs: 字典，将单词（字符串）映射为它们在训练期间所出现的文档或文本的数量。仅在调用fit_on_texts之后设置。
- word_index: 字典，将单词（字符串）映射为它们的排名或者索引。仅在调用fit_on_texts之后设置。
- document_count: 整数。分词器被训练的文档（文本或者序列）数量。仅在调用fit_on_texts或fit_on_sequences之后设置。

图片预处理

图片生成器ImageDataGenerator

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=0.,
    width_shift_range=0.,
    height_shift_range=0.,
    shear_range=0.,
    zoom_range=0.,
    channel_shift_range=0.,
    fill_mode='nearest',
    cval=0.,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    preprocessing_function=None,
    data_format=K.image_data_format())
```

用以生成一个batch的图像数据，支持实时数据提升。训练时该函数会无限生成数据，直到达到规定的epoch次数为止。

参数

- featurewise_center：布尔值，使输入数据集去中心化（均值为0）, 按feature执行
- samplewise_center：布尔值，使输入数据的每个样本均值为0
- featurewise_std_normalization：布尔值，将输入除以数据集的标准差以完成标准化, 按feature执行

- samplewise_std_normalization：布尔值，将输入的每个样本除以其自身的标准差
- zca_whitening：布尔值，对输入数据施加ZCA白化
- rotation_range：整数，数据提升时图片随机转动的角度
- width_shift_range：浮点数，图片宽度的某个比例，数据提升时图片水平偏移的幅度
- height_shift_range：浮点数，图片高度的某个比例，数据提升时图片竖直偏移的幅度
- shear_range：浮点数，剪切强度（逆时针方向的剪切变换角度）
- zoom_range：浮点数或形如 `[lower,upper]` 的列表，随机缩放的幅度，若为浮点数，则相当于 `[lower,upper] = [1 - zoom_range, 1+zoom_range]`
- channel_shift_range：浮点数，随机通道偏移的幅度
- fill_mode：；‘constant’，‘nearest’，‘reflect’或‘wrap’之一，当进行变换时超出边界的点将根据本参数给定的方法进行处理
- cval：浮点数或整数，当 `fill_mode=constant` 时，指定要向超出边界的点填充的值
- horizontal_flip：布尔值，进行随机水平翻转
- vertical_flip：布尔值，进行随机竖直翻转
- rescale: 重放缩因子,默认为None. 如果为None或0则不进行放缩,否则会将该数值乘到数据上(在应用其他变换之前)
- preprocessing_function: 将被应用于每个输入的函数。该函数将在任何其他修改之前运行。该函数接受一个参数，为一张图片（秩为3的numpy array），并且输出一个具有相同shape的numpy array
- data_format：字符串，“channel_first”或“channel_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image_dim_ordering，“channel_last”对应原本的“tf”，“channel_first”对应原本的“th”。以128x128的RGB图像为

例，“channel_first”应将数据组织为（3,128,128），而“channel_last”应将数据组织为（128,128,3）。该参数的默认值是 `~/.keras/keras.json` 中设置的值，若从未设置过，则为“channel_last”

方法

- fit(x, augment=False, rounds=1)：计算依赖于数据的变换所需要的统计信息(均值方差等),只有使用 `featurewise_center`，`featurewise_std_normalization` 或 `zca_whitening` 时需要此函数。
  - X：numpy array，样本数据，秩应为4.在黑白图像的情况下channel轴的值为1，在彩色图像情况下值为3
  - augment：布尔值，确定是否使用随即提升过的数据
  - round：若设 `augment=True`，确定要在数据上进行多少轮数据提升，默认值为1
  - seed: 整数,随机数种子
- flow(self, X, y, batch_size=32, shuffle=True, seed=None, save_to_dir=None, save_prefix='', save_format='jpeg')：接收numpy数组和标签为参数,生成经过数据提升或标准化后的batch数据,并在一个无限循环中不断的返回batch数据
  - x：样本数据，秩应为4.在黑白图像的情况下channel轴的值为1，在彩色图像情况下值为3
  - y：标签
  - batch_size：整数，默认32
  - shuffle：布尔值，是否随机打乱数据，默认为True
  - save_to_dir：None或字符串，该参数能让你将提升后的图片保存起来，用以可视化
  - save_prefix：字符串，保存提升后图片时使用的前缀, 仅当设置了 `save_to_dir` 时生效

- save_format："png"或"jpeg"之一，指定保存图片的数据格式,默认"jpeg"
- yields:形如(x,y)的tuple,x是代表图像数据的numpy数组.y是代表标签的numpy数组.该迭代器无限循环.
- seed: 整数,随机数种子

* flow_from_directory(directory): 以文件夹路径为参数,生成经过数据提升/归一化后的数据,在一个无限循环中无限产生batch数据
  - directory: 目标文件夹路径,对于每一个类,该文件夹都要包含一个子文件夹.子文件夹中任何JPG、PNG和BNP的图片都会被生成器使用.详情请查看此脚本
  - target_size: 整数tuple,默认为(256, 256). 图像将被resize成该尺寸
  - color_mode: 颜色模式,为"grayscale","rgb"之一,默认为"rgb".代表这些图片是否会被转换为单通道或三通道的图片.
  - classes: 可选参数,为子文件夹的列表,如['dogs','cats']默认为None. 若未提供,则该类别列表将从 `directory` 下的子文件夹名称/结构自动推断。每一个子文件夹都会被认为是一个新的类。(类别的顺序将按照字母表顺序映射到标签值)。通过属性 `class_indices` 可获得文件夹名与类的序号的对应字典。
  - class_mode: "categorical", "binary", "sparse"或None之一. 默认为"categorical. 该参数决定了返回的标签数组的形式, "categorical"会返回2D的one-hot编码标签,"binary"返回1D的二值标签."sparse"返回1D的整数标签,如果为None则不返回任何标签, 生成器将仅仅生成batch数据, 这种情况在使用 `model.predict_generator()` 和 `model.evaluate_generator()` 等函数时会用到.
  - batch_size: batch数据的大小,默认32
  - shuffle: 是否打乱数据,默认为True
  - seed: 可选参数,打乱数据和进行变换时的随机数种子
  - save_to_dir: None或字符串，该参数能让你将提升后的图片保存起来，用以可视化
  - save_prefix：字符串，保存提升后图片时使用的前缀, 仅当设置了 `save_to_dir` 时生效
  - save_format："png"或"jpeg"之一，指定保存图片的数据格式,默认"jpeg"
  - flollow_links: 是否访问子文件夹中的软链接

例子

使用 `.flow()` 的例子

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is appli
ed)
datagen.fit(x_train)

# fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(x_train, y_train, batch_size=32
),
                    steps_per_epoch=len(x_train), epochs=epochs)

# here's a more "manual" example
for e in range(epochs):
    print 'Epoch', e
    batches = 0
    for x_batch, y_batch in datagen.flow(x_train, y_train, batch
_size=32):
        loss = model.train(x_batch, y_batch)
        batches += 1
        if batches >= len(x_train) / 32:
            # we need to break the loop by hand because
            # the generator loops indefinitely
            break
```

使用 `.flow_from_directory(directory)` 的例子

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

model.fit_generator(
        train_generator,
        steps_per_epoch=2000,
        epochs=50,
        validation_data=validation_generator,
        validation_steps=800)
```

同时变换图像和mask

```
# we create two instances with the same arguments
data_gen_args = dict(featurewise_center=True,
                     featurewise_std_normalization=True,
                     rotation_range=90.,
                     width_shift_range=0.1,
                     height_shift_range=0.1,
                     zoom_range=0.2)
image_datagen = ImageDataGenerator(**data_gen_args)
mask_datagen = ImageDataGenerator(**data_gen_args)

# Provide the same seed and keyword arguments to the fit and flo
w methods
seed = 1
image_datagen.fit(images, augment=True, seed=seed)
mask_datagen.fit(masks, augment=True, seed=seed)

image_generator = image_datagen.flow_from_directory(
    'data/images',
    class_mode=None,
    seed=seed)

mask_generator = mask_datagen.flow_from_directory(
    'data/masks',
    class_mode=None,
    seed=seed)

# combine generators into one which yields image and masks
train_generator = zip(image_generator, mask_generator)

model.fit_generator(
    train_generator,
    steps_per_epoch=2000,
    epochs=50)
```

目标函数objectives

目标函数，或称损失函数，是编译一个模型必须的两个参数之一：

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

可以通过传递预定义目标函数名字指定目标函数，也可以传递一个Theano/TensroFlow的符号函数作为目标函数，该函数对每个数据点应该只返回一个标量值，并以下列两个参数为参数：

- y_true：真实的数据标签，Theano/TensorFlow张量
- y_pred：预测值，与y_true相同shape的Theano/TensorFlow张量 ```python from keras import losses

model.compile(loss=losses.mean_squared_error, optimizer='sgd')

```
真实的优化目标函数是在各个数据点得到的损失函数值之和的均值

请参考[目标实现代码](https://github.com/fchollet/keras/blob/master/
keras/objectives.py)获取更多信息

## 可用的目标函数

* mean_squared_error或mse

* mean_absolute_error或mae

* mean_absolute_percentage_error或mape

* mean_squared_logarithmic_error或msle

* squared_hinge

* hinge

* binary_crossentropy（亦称作对数损失，logloss）

* logcosh

* categorical_crossentropy：亦称作多类的对数损失，注意使用该目标函数时，
需要将标签转化为形如```(nb_samples, nb_classes)```的二值序列

* sparse_categorical_crossentrop：如上，但接受稀疏标签。注意，使用该函
数时仍然需要你的标签与输出值的维度相同，你可能需要在标签数据上增加一个维度：`
``np.expand_dims(y,-1)
```

- kullback_leibler_divergence:从预测值概率分布Q到真值概率分布P的信息增益,用以度量两个分布的差异.
- poisson：即 `(predictions - targets * log(predictions))` 的均值
- cosine_proximity：即预测值与真实标签的余弦距离平均值的相反数

注意: 当使用"categorical_crossentropy"作为目标函数时,标签应该为多类模式,即one-hot编码的向量,而不是单个数值. 可以使用工具中的 `to_categorical` 函数完成该转换.示例如下:

```
from keras.utils.np_utils import to_categorical

categorical_labels = to_categorical(int_labels, num_classes=None
)
```

优化器optimizers

优化器是编译Keras模型必要的两个参数之一

```
from keras import optimizers

model = Sequential()
model.add(Dense(64, init='uniform', input_shape=(10,)))
model.add(Activation('tanh'))
model.add(Activation('softmax'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=
True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

可以在调用 `model.compile()` 之前初始化一个优化器对象，然后传入该函数（如上所示），也可以在调用 `model.compile()` 时传递一个预定义优化器名。在后者情形下，优化器的参数将使用默认值。

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```

所有优化器都可用的参数

参数 `clipnorm` 和 `clipvalue` 是所有优化器都可以使用的参数,用于对梯度进行裁剪.示例如下:

```
from keras import optimizers

# All parameter gradients will be clipped to
# a maximum norm of 1.
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
```

```
from keras import optimizers

# All parameter gradients will be clipped to
# a maximum value of 0.5 and
# a minimum value of -0.5.
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)
```

SGD

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=
False)
```

随机梯度下降法，支持动量参数，支持学习衰减率，支持Nesterov动量

参数

- lr：大于0的浮点数，学习率
- momentum：大于0的浮点数，动量参数
- decay：大于0的浮点数，每次更新后的学习率衰减值
- nesterov：布尔值，确定是否使用Nesterov动量

RMSprop

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)
```

除学习率可调整外，建议保持优化器的其他默认参数不变

该优化器通常是面对递归神经网络时的一个良好选择

参数

- lr：大于0的浮点数，学习率
- rho：大于0的浮点数
- epsilon：大于0的小浮点数，防止除0错误

Adagrad

```
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-06)
```

建议保持优化器的默认参数不变

Adagrad

- lr：大于0的浮点数，学习率
- epsilon：大于0的小浮点数，防止除0错误

Adadelta

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-06)
```

建议保持优化器的默认参数不变

参数

- lr：大于0的浮点数，学习率
- rho：大于0的浮点数
- epsilon：大于0的小浮点数，防止除0错误

参考文献

---

* Adadelta - an adaptive learning rate method

Adam

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

该优化器的默认值来源于参考文献

参数

* lr：大于0的浮点数，学习率
* beta_1/beta_2：浮点数， 0<beta<1，通常很接近1
* epsilon：大于0的小浮点数，防止除0错误

参考文献

* Adam - A Method for Stochastic Optimization

Adamax

```
keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

Adamax优化器来自于Adam的论文的Section7，该方法是基于无穷范数的Adam方法的变体。

默认参数由论文提供

参数

- lr：大于0的浮点数，学习率
- beta_1/beta_2：浮点数， 0<beta<1，通常很接近1
- epsilon：大于0的小浮点数，防止除0错误

参考文献

- Adam - A Method for Stochastic Optimization

Nadam

```
keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08, schedule_decay=0.004)
```

Nesterov Adam optimizer: Adam本质上像是带有动量项的RMSprop，Nadam就是带有Nesterov 动量的Adam RMSprop

默认参数来自于论文，推荐不要对默认参数进行更改。

参数

- lr：大于0的浮点数，学习率
- beta_1/beta_2：浮点数， 0<beta<1，通常很接近1
- epsilon：大于0的小浮点数，防止除0错误

参考文献

- Nadam report
- On the importance of initialization and momentum in deep learning

TFOptimizer

```
keras.optimizers.TFOptimizer(optimizer)
```

TF优化器的包装器

激活函数Activations

激活函数可以通过设置单独的激活层实现，也可以在构造层对象时通过传递 `activation` 参数实现。

```
from keras.layers import Activation, Dense

model.add(Dense(64))
model.add(Activation('tanh'))
```

等价于

```
model.add(Dense(64, activation='tanh'))
```

也可以通过传递一个逐元素运算的Theano/TensorFlow函数来作为激活函数：

```
from keras import backend as K

def tanh(x):
    return K.tanh(x)

model.add(Dense(64, activation=tanh))
model.add(Activation(tanh)
```

预定义激活函数

- softmax：对输入数据的最后一维进行softmax，输入数据应形如 `(nb_samples, nb_timesteps, nb_dims)` 或 `(nb_samples,nb_dims)`
- elu
- softplus
- softsign

- relu
- tanh
- sigmoid
- hard_sigmoid
- linear

高级激活函数

对于简单的Theano/TensorFlow不能表达的复杂激活函数，如含有可学习参数的激活函数，可通过高级激活函数实现，如PReLU，LeakyReLU等

性能评估

使用方法

性能评估模块提供了一系列用于模型性能评估的函数,这些函数在模型编译时由 `metrices` 关键字设置

性能评估函数类似与目标函数, 只不过该性能的评估结果讲不会用于训练.

可以通过字符串来使用域定义的性能评估函数

```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['mae', 'acc'])
```

也可以自定义一个Theano/TensorFlow函数并使用之

```
from keras import metrics

model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=[metrics.mae, metrics.categorical_accuracy
])
```

参数

- y_true:真实标签,theano/tensorflow张量
- y_pred:预测值, 与y_true形式相同的theano/tensorflow张量

返回值

单个用以代表输出各个数据点上均值的值

可用预定义张量

除fbeta_score额外拥有默认参数beta=1外,其他各个性能指标的参数均为y_true和y_pred

- binary_accuracy: 对二分类问题,计算在所有预测值上的平均正确率
- categorical_accuracy:对多分类问题,计算再所有预测值上的平均正确率
- sparse_categorical_accuracy:与 `categorical_accuracy` 相同,在对稀疏的目标值预测时有用
- top_k_categorical_accracy: 计算top-k正确率,当预测值的前k个值中存在目标类别即认为预测正确

定制评估函数

定制的评估函数可以在模型编译时传入,该函数应该以 `(y_true, y_pred)` 为参数,并返回单个张量,或从 `metric_name` 映射到 `metric_value` 的字典,下面是一个示例:

```
(y_true, y_pred) as arguments and return a single tensor value.

import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

初始化方法

初始化方法定义了对Keras层设置初始化权重的方法

不同的层可能使用不同的关键字来传递初始化方法，一般来说指定初始化方法的关键字是 `kernel_initializer` 和 `bias_initializer` ，例如：

```
model.add(Dense(64,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
```

一个初始化器可以由字符串指定（必须是下面的预定义初始化器之一），或一个callable的函数，例如

```
from keras import initializers

model.add(Dense(64, kernel_initializer=initializers.random_normal(stddev=0.01)))

# also works; will use the default parameters.
model.add(Dense(64, kernel_initializer='random_normal'))
```

Initializer

Initializer是所有初始化方法的父类，不能直接使用，如果想要定义自己的初始化方法，请继承此类。

预定义初始化方法

Zeros

```
keras.initializers.Zeros()
```

全零初始化

Ones

```
keras.initializers.Ones()
```

全1初始化

Constant

```
keras.initializers.Constant(value=0)
```

初始化为固定值value

RandomNormal

```
keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None
))
```

正态分布初始化

- mean：均值
- stddev：标准差
- seed：随机数种子

RandomUniform

```
keras.initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=
None)
```

均匀分布初始化

- minval：均匀分布下边界

- maxval：均匀分布上边界
- seed：随机数种子

TruncatedNormal

```
keras.initializers.TruncatedNormal(mean=0.0, stddev=0.05, seed=None)
```

截尾高斯分布初始化，该初始化方法与RandomNormal类似，但位于均值两个标准差以外的数据将会被丢弃并重新生成，形成截尾分布。该分布是神经网络权重和滤波器的推荐初始化方法。

- mean：均值
- stddev：标准差
- seed：随机数种子

VarianceScaling

```
keras.initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='normal', seed=None)
```

该初始化方法能够自适应目标张量的shape。

当 `distribution="normal"` 时，样本从0均值，标准差为sqrt(scale / n)的截尾正态分布中产生。其中：

```
* 当```mode = "fan_in"```时，权重张量的输入单元数。
* 当```mode = "fan_out"```时，权重张量的输出单元数
* 当```mode = "fan_avg"```时，权重张量的输入输出单元数的均值
```

当 `distribution="uniform"` 时，权重从[-limit, limit]范围内均匀采样，其中limit = limit = sqrt(3 * scale / n)

- scale: 放缩因子，正浮点数
- mode: 字符串，“fan_in”，“fan_out”或“fan_avg”fan_in", "fan_out", "fan_avg".
- distribution: 字符串，“normal”或“uniform”.

- seed: 随机数种子

Orthogonal

```
keras.initializers.Orthogonal(gain=1.0, seed=None)
```

用随机正交矩阵初始化

- gain: 正交矩阵的乘性系数
- seed：随机数种子

参考文献：Saxe et al.

Identiy

```
keras.initializers.Identity(gain=1.0)
```

使用单位矩阵初始化，仅适用于2D方阵

- gain：单位矩阵的乘性系数

lecun_uniform

```
lecun_uniform(seed=None)
```

LeCun均匀分布初始化方法，参数由[-limit, limit]的区间中均匀采样获得，其中limit=sqrt(3 / fan_in), fin_in是权重向量的输入单元数（扇入）

- seed：随机数种子

参考文献：LeCun 98, Efficient Backprop

glorot_normal

```
glorot_normal(seed=None)
```

Glorot正态分布初始化方法，也称作Xavier正态分布初始化，参数由0均值，标准差为sqrt(2 / (fan_in + fan_out))的正态分布产生，其中fan_in和fan_out是权重张量的扇入扇出（即输入和输出单元数目）

- seed：随机数种子

参考文献：Glorot & Bengio, AISTATS 2010

he_normal

```
he_normal(seed=None)
```

He正态分布初始化方法，也称作Xavier正态分布初始化，参数由0均值，标准差为sqrt(2 / fan_in) 的正态分布产生，其中fan_in权重张量的扇入

- seed：随机数种子

参考文献：He et al

he_uniform

```
he_normal(seed=None)
```

LeCun均匀分布初始化方法，参数由[-limit, limit]的区间中均匀采样获得，其中limit=sqrt(6 / fan_in), fin_in是权重向量的输入单元数（扇入）

- seed：随机数种子

参考文献：He et al

自定义初始化器

如果需要传递自定义的初始化器，则该初始化器必须是callable的，并且接收 `shape` （将被初始化的张量shape）和 `dtype` （数据类型）两个参数，并返回符合 `shape` 和 `dtype` 的张量。

```python
from keras import backend as K

def my_init(shape, dtype=None):
    return K.random_normal(shape, dtype=dtype)

model.add(Dense(64, init=my_init))
```

正则项

正则项在优化过程中层的参数或层的激活值添加惩罚项，这些惩罚项将与损失函数一起作为网络的最终优化目标

惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但 `Dense, Conv1D, Conv2D, Conv3D` 具有共同的接口。

这些层有三个关键字参数以施加正则项：

- `kernel_regularizer`：施加在权重上的正则项，为 `keras.regularizer.Regularizer` 对象
- `bias_regularizer`：施加在偏置向量上的正则项，为 `keras.regularizer.Regularizer` 对象
- `activity_regularizer`：施加在输出上的正则项，为 `keras.regularizer.Regularizer` 对象

例子

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

可用正则项

```
keras.regularizers.l1(0.)
keras.regularizers.l2(0.)
keras.regularizers.l1_l2(0.)
```

开发新的正则项

任何以权重矩阵作为输入并返回单个数值的函数均可以作为正则项，示例：

```
from keras import backend as K

def l1_reg(weight_matrix):
    return 0.01 * K.sum(K.abs(weight_matrix))

model.add(Dense(64, input_dim=64,
                kernel_regularizer=l1_reg)
```

可参考源代码keras/regularizer.py

约束项

来自 `constraints` 模块的函数在优化过程中为网络的参数施加约束

惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但 `Dense, Conv1D, Conv2D, Conv3D` 具有共同的接口。

这些层通过一下关键字施加约束项

- `kernel_constraint` ：对主权重矩阵进行约束
- `bias_constraint` ：对偏置向量进行约束

```
from keras.constraints import maxnorm
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```

预定义约束项

- max_norm(m=2)：最大模约束
- non_neg()：非负性约束
- unit_norm()：单位范数约束, 强制矩阵沿最后一个轴拥有单位范数

回调函数Callbacks

回调函数是一组在训练的特定阶段被调用的函数集，你可以使用回调函数来观察训练过程中网络内部的状态和统计信息。通过传递回调函数列表到模型的 `.fit()` 中，即可在给定的训练阶段调用该函数集中的函数。

【Tips】虽然我们称之为回调“函数”，但事实上Keras的回调函数是一个类，回调函数只是习惯性称呼

Callback

```
keras.callbacks.Callback()
```

这是回调函数的抽象类，定义新的回调函数必须继承自该类

类属性

- params：字典，训练参数集（如信息显示方法verbosity，batch大小，epoch数）
- model： `keras.models.Model` 对象，为正在训练的模型的引用

回调函数以字典 `logs` 为参数，该字典包含了一系列与当前batch或epoch相关的信息。

目前，模型的 `.fit()` 中有下列参数会被记录到 `logs` 中：

- 在每个epoch的结尾处（on_epoch_end）， `logs` 将包含训练的正确率和误差， `acc` 和 `loss` ，如果指定了验证集，还会包含验证集正确率和误差 `val_acc)` 和 `val_loss` ， `val_acc` 还额外需要在 `.compile` 中启用 `metrics=['accuracy']` 。
- 在每个batch的开始处（on_batch_begin）： `logs` 包含 `size` ，即当前batch的样本数

- 在每个batch的结尾处（on_batch_end）： `logs` 包含 `loss` ，若启用 `accuracy` 则还包含 `acc`

BaseLogger

```
keras.callbacks.BaseLogger()
```

该回调函数用来对每个epoch累加 `metrics` 指定的监视指标的epoch平均值

该回调函数在每个Keras模型中都会被自动调用

ProgbarLogger

```
keras.callbacks.ProgbarLogger()
```

该回调函数用来将 `metrics` 指定的监视指标输出到标准输出上

History

```
keras.callbacks.History()
```

该回调函数在Keras模型上会被自动调用， `History` 对象即为 `fit` 方法的返回值

ModelCheckpoint

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False, save_weights_only=False, mode='auto', period=1)
```

该回调函数将在每个epoch后保存模型到 `filepath`

```


例如，```filepath```若为```weights.{epoch:02d-{val_loss:.2f}}.hdf5```，则会生成对应epoch和验证集loss的多个文件。


### 参数


* filename：字符串，保存模型的路径


* monitor：需要监视的值


* verbose：信息展示模式，0或1


* save_best_only：当设置为```True```时，将只保存在验证集上性能最好的模型


* mode：‘auto’，‘min’，‘max’之一，在```save_best_only=True```时决定性能最佳模型的评判准则，例如，当监测值为```val_acc```时，模式应为```max```，当检测值为```val_loss```时，模式应为```min```。在```auto```模式下，评价准则由被监测值的名字自动推断。


* save_weights_only：若设置为True，则只保存模型权重，否则将保存整个模型（包括模型结构，配置信息等）


* period：CheckPoint之间的间隔的epoch数


***


## EarlyStopping
```python
keras.callbacks.EarlyStopping(monitor='val_loss', patience=0, verbose=0, mode='auto')
```

当监测值不再改善时，该回调函数将中止训练
```

参数

- monitor：需要监视的量
- patience：当early stop被激活（如发现loss相比上一个epoch训练没有下降），则经过 `patience` 个epoch后停止训练。
- verbose：信息展示模式
- mode：‘auto’，‘min’，‘max’之一，在 `min` 模式下，如果检测值停止下降则中止训练。在 `max` 模式下，当检测值不再上升则停止训练。

RemoteMonitor

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000')
```

该回调函数用于向服务器发送事件流，该回调函数需要 `requests` 库

参数

- root：该参数为根url，回调函数将在每个epoch后把产生的事件流发送到该地址，事件将被发往 `root + '/publish/epoch/end/'` 。发送方法为HTTP POST，其 `data` 字段的数据是按JSON格式编码的事件字典。

LearningRateScheduler

```
keras.callbacks.LearningRateScheduler(schedule)
```

该回调函数是学习率调度器

参数

- schedule：函数，该函数以epoch号为参数（从0算起的整数），返回一个新学

习率（浮点数）

TensorBoard

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0,
write_graph=True, write_images=False, embeddings_freq=0, embeddi
ngs_layer_names=None, embeddings_metadata=None)
```

该回调函数是一个可视化的展示器

TensorBoard是TensorFlow提供的可视化工具，该回调函数将日志信息写入TensorBorad，使得你可以动态的观察训练和测试指标的图像以及不同层的激活值直方图。

如果已经通过pip安装了TensorFlow，我们可通过下面的命令启动TensorBoard：

```
tensorboard --logdir=/full_path_to_your_logs
```

更多的参考信息，请点击这里

参数

- log_dir：保存日志文件的地址，该文件将被TensorBoard解析以用于可视化
- histogram_freq：计算各个层激活值直方图的频率（每多少个epoch计算一次），如果设置为0则不计算。
- write_graph: 是否在Tensorboard上可视化图，当设为True时，log文件可能会很大
- write_images: 是否将模型权重以图片的形式可视化
- embeddings_freq: 依据该频率(以epoch为单位)筛选保存的embedding层
- embeddings_layer_names:要观察的层名称的列表，若设置为None或空列表，则所有embedding层都将被观察。
- embeddings*metadata: 字典，将层名称映射为包含该embedding*层元数据的文件名，参考*[这里]*

(https://keras.io/https_://www.tensorflow.org/how_tos/embedding_viz/#metadata_optional)获得元数据文件格式的细节。如果所有的embedding层都使用相同的元数据文件，则可传递字符串。

ReduceLROnPlateau

```
keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.1
, patience=10, verbose=0, mode='auto', epsilon=0.0001, cooldown=0
, min_lr=0)
```

当评价指标不在提升时，减少学习率

当学习停滞时，减少2倍或10倍的学习率常常能获得较好的效果。该回调函数检测指标的情况，如果在 `patience` 个epoch中看不到模型性能提升，则减少学习率

参数

- monitor：被监测的量
- factor：每次减少学习率的因子，学习率将以 `lr = lr*factor` 的形式被减少
- patience：当patience个epoch过去而模型性能不提升时，学习率减少的动作会被触发
- mode：'auto'，'min'，'max'之一，在 `min` 模式下，如果检测值触发学习率减少。在 `max` 模式下，当检测值不再上升则触发学习率减少。
- epsilon：阈值，用来确定是否进入检测值的“平原区”
- cooldown：学习率减少后，会经过cooldown个epoch才重新进行正常操作
- min_lr：学习率的下限

CSVLogger

```
keras.callbacks.CSVLogger(filename, separator=',', append=False)
```

将epoch的训练结果保存在csv文件中，支持所有可被转换为string的值，包括1D的可迭代数值如np.ndarray.

参数

- fiename：保存的csv文件名，如 `run/log.csv`
- separator：字符串，csv分隔符
- append：默认为False，为True时csv文件如果存在则继续写入，为False时总是覆盖csv文件

LambdaCallback

```
keras.callbacks.LambdaCallback(on_epoch_begin=None, on_epoch_end=None, on_batch_begin=None, on_batch_end=None, on_train_begin=None, on_train_end=None)
```

用于创建简单的callback的callback类

该callback的匿名函数将会在适当的时候调用，注意，该回调函数假定了一些位置参数 `on_eopoch_begin` 和 `on_epoch_end` 假定输入的参数是 `epoch, logs` . `on_batch_begin` 和 `on_batch_end` 假定输入的参数是 `batch, logs` ， `on_train_begin` 和 `on_train_end` 假定输入的参数是 `logs`

参数

- on_epoch_begin: 在每个epoch开始时调用
- on_epoch_end: 在每个epoch结束时调用
- on_batch_begin: 在每个batch开始时调用
- on_batch_end: 在每个batch结束时调用
- on_train_begin: 在训练开始时调用
- on_train_end: 在训练结束时调用

示例

```
# Print the batch number at the beginning of every batch.
batch_print_callback = LambdaCallback(
    on_batch_begin=lambda batch,logs: print(batch))

# Plot the loss after every epoch.
import numpy as np
import matplotlib.pyplot as plt
plot_loss_callback = LambdaCallback(
    on_epoch_end=lambda epoch, logs: plt.plot(np.arange(epoch),
                      logs['loss']))

# Terminate some processes after having finished model training.
processes = ...
cleanup_callback = LambdaCallback(
    on_train_end=lambda logs: [
    p.terminate() for p in processes if p.is_alive()])

model.fit(...,
      callbacks=[batch_print_callback,
         plot_loss_callback,
         cleanup_callback])
```

编写自己的回调函数

我们可以通过继承 `keras.callbacks.Callback` 编写自己的回调函数，回调函数通过类成员 `self.model` 访问访问，该成员是模型的一个引用。

这里是一个简单的保存每个batch的loss的回调函数：

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
```

例子：记录损失函数的历史数据

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform')
)
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmspro
p')

history = LossHistory()
model.fit(X_train, Y_train, batch_size=128, epochs=20, verbose=0
, callbacks=[history])

print history.losses
# outputs
'''
[0.66047596406559383, 0.3547245744908703, ..., 0.259531552041596
17, 0.25901699725311789]
```

例子：模型检查点

```
from keras.callbacks import ModelCheckpoint

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform')
)
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmspro
p')

'''
saves the model weights after each epoch if the validation loss
decreased
'''
checkpointer = ModelCheckpoint(filepath="/tmp/weights.hdf5", ver
bose=1, save_best_only=True)
model.fit(X_train, Y_train, batch_size=128, epochs=20, verbose=0
, validation_data=(X_test, Y_test), callbacks=[checkpointer])
```

Э��ʹ��Keras

Application应用

Kera的应用模块Application提供了带有预训练权重的Keras模型，这些模型可以用来进行预测、特征提取和finetune

模型的预训练权重将下载到 `~/.keras/models/` 并在载入模型时自动载入

可用的模型

应用于图像分类的模型,权重训练自ImageNet：

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3

所有的这些模型(除了Xception)都兼容Theano和Tensorflow，并会自动基于 `~/.keras/keras.json` 的Keras的图像维度进行自动设置。例如，如果你设置 `data_format="channel_last"` ，则加载的模型将按照TensorFlow的维度顺序来构造，即“Width-Height-Depth”的顺序

图片分类模型的示例

利用ResNet50网络进行ImageNet分类

```
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode
_predictions
import numpy as np

model = ResNet50(weights='imagenet')

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
# decode the results into a list of tuples (class, description,
probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
# Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u
'n01871265', u'tusker', 0.1122357), (u'n02504458', u'African_ele
phant', 0.061040461)]
```

利用VGG16提取特征

```
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

model = VGG16(weights='imagenet', include_top=False)

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

features = model.predict(x)
```

从VGG19的任意中间层中抽取特征

```
from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing import image
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D
from keras import backend as K

# create the base pre-trained model
base_model = InceptionV3(weights='imagenet', include_top=False)

# add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)
# let's add a fully-connected layer
x = Dense(1024, activation='relu')(x)
# and a logistic layer -- let's say we have 200 classes
predictions = Dense(200, activation='softmax')(x)

# this is the model we will train
model = Model(inputs=base_model.input, outputs=predictions)

# first: train only the top layers (which were randomly initiali
```

```
zed)
# i.e. freeze all convolutional InceptionV3 layers
for layer in base_model.layers:
    layer.trainable = False

# compile the model (should be done *after* setting layers to no
n-trainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentrop
y')

# train the model on the new data for a few epochs
model.fit_generator(...)

# at this point, the top layers are well trained and we can star
t fine-tuning
# convolutional layers from inception V3. We will freeze the bot
tom N layers
# and train the remaining top layers.

# let's visualize layer names and layer indices to see how many
layers
# we should freeze:
for i, layer in enumerate(base_model.layers):
   print(i, layer.name)

# we chose to train the top 2 inception blocks, i.e. we will fre
eze
# the first 172 layers and unfreeze the rest:
for layer in model.layers[:172]:
   layer.trainable = False
for layer in model.layers[172:]:
   layer.trainable = True

# we need to recompile the model for these modifications to take
 effect
# we use SGD with a low learning rate
from keras.optimizers import SGD
model.compile(optimizer=SGD(lr=0.0001, momentum=0.9), loss='cate
gorical_crossentropy')
```

```
# we train our model again (this time fine-tuning the top 2 ince
ption blocks
# alongside the top Dense layers
model.fit_generator(...)
```

在定制的输入tensor上构建InceptionV3

```
from keras.applications.inception_v3 import InceptionV3
from keras.layers import Input

# this could also be the output a different Keras model or layer
input_tensor = Input(shape=(224, 224, 3))  # this assumes K.imag
e_data_format() == 'channels_last'

model = InceptionV3(input_tensor=input_tensor, weights='imagenet'
, include_top=True)
```

模型文档

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- MusicTaggerCRNN

Xception模型


```
keras.applications.xception.Xception(include_top=True, weights='imagenet',
                                      input_tensor=None, input_shape=None,
                                      pooling=None, classes=1000)
```

Xception V1 模型, 权重由ImageNet训练而言

在ImageNet上,该模型取得了验证集top1 0.790和top5 0.945的正确率

注意,该模型目前仅能以TensorFlow为后端使用,由于它依赖于"SeparableConvolution"层,目前该模型只支持channels_last的维度顺序(width, height, channels)

默认输入图片大小为299x299

参数

- include_top:是否保留顶层的3个全连接网络
- weights:None代表随机初始化,即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor:可填入Keras tensor作为模型的图像输出tensor
- input_shape:可选,仅当 `include_top=False` 有效,应为长为3的tuple,指明输入图片的shape,图片的宽高必须大于71,如(150,150,3)
- pooling:当include_top=False时,该参数指定了池化方式。None代表不池化,最后一个卷积层的输出为4D张量。'avg'代表全局平均池化,'max'代表全局最大值池化。
- classes:可选,图片分类的类别数,仅当 `include_top=True` 并且不加载预训练权重时可用。

返回值

Keras 模型对象

参考文献

- Xception: Deep Learning with Depthwise Separable Convolutions

License

预训练权重由我们自己训练而来，基于MIT license发布

VGG16模型


```
keras.applications.vgg16.VGG16(include_top=True, weights='imagenet',
                                input_tensor=None, input_shape=None,
                                pooling=None,
                                classes=1000)
```

VGG16模型,权重由ImageNet训练而来

该模型再Theano和TensorFlow后端均可使用,并接受channels_first和channels_last两种输入维度顺序

模型的默认输入尺寸时224x224

参数

- include_top：是否保留顶层的3个全连接网络
- weights：None代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入Keras tensor作为模型的图像输出tensor
- input_shape：可选，仅当 `include_top=False` 有效，应为长为3的tuple，指明输入图片的shape，图片的宽高必须大于48，如(200,200,3)

返回值

- pooling：当include_top=False时，该参数指定了池化方式。None代表不池化，最后一个卷积层的输出为4D张量。‘avg’代表全局平均池化，‘max’代表全局最大值池化。
- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

Keras 模型对象

参考文献

- Very Deep Convolutional Networks for Large-Scale Image Recognition：如果在研究中使用了VGG，请引用该文

License

VGG19模型


```
keras.applications.vgg19.VGG19(include_top=True, weights='imagenet',
                                input_tensor=None, input_shape=None,
                                pooling=None,
                                classes=1000)
```

VGG19模型,权重由ImageNet训练而来

该模型在Theano和TensorFlow后端均可使用,并接受channels_first和channels_last两种输入维度顺序

模型的默认输入尺寸时224x224

参数

- include_top：是否保留顶层的3个全连接网络
- weights：None代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入Keras tensor作为模型的图像输出tensor
- input_shape：可选，仅当`include_top=False`有效，应为长为3的tuple，指明输入图片的shape，图片的宽高必须大于48，如(200,200,3)
- pooling：当include_top=False时，该参数指定了池化方式。None代表不池化，最后一个卷积层的输出为4D张量。'avg'代表全局平均池化，'max'代表全局最大值池化。
- classes：可选，图片分类的类别数，仅当`include_top=True`并且不加载预训练权重时可用。

返回值

返回值

Keras 模型对象

参考文献

- Very Deep Convolutional Networks for Large-Scale Image Recognition：如果在研究中使用了VGG，请引用该文

License

预训练权重由牛津VGG组发布的预训练权重移植而来，基于Creative Commons Attribution License

---

ResNet50模型


```
keras.applications.resnet50.ResNet50(include_top=True, weights='imagenet',
                                     input_tensor=None, input_shape=None,
                                     pooling=None,
                                     classes=1000)
```

50层残差网络模型,权重训练自ImageNet

该模型在Theano和TensorFlow后端均可使用,并接受channels_first和channels_last两种输入维度顺序

模型的默认输入尺寸时224x224

参数

- include_top：是否保留顶层的全连接网络
- weights：None代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入Keras tensor作为模型的图像输出tensor
- input_shape：可选，仅当 `include_top=False` 有效，应为长为3的tuple，指明输入图片的shape，图片的宽高必须大于197，如(200,200,3)
- pooling：当include_top=False时，该参数指定了池化方式。None代表不池化，最后一个卷积层的输出为4D张量。‘avg’代表全局平均池化，‘max’代表全局最大值池化。
- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

返回值

Keras 模型对象

参考文献

- Deep Residual Learning for Image Recognition：如果在研究中使用了ResNet50，请引用该文

License

预训练权重由Kaiming He发布的预训练权重移植而来，基于MIT License

InceptionV3模型


```
keras.applications.inception_v3.InceptionV3(include_top=True,
                                            weights='imagenet',
                                            input_tensor=None,
                                            input_shape=None,
                                            pooling=None,
                                            classes=1000)
```

InceptionV3网络,权重训练自ImageNet

该模型在Theano和TensorFlow后端均可使用,并接受channels_first和channels_last两种输入维度顺序

模型的默认输入尺寸时299x299

参数

- include_top：是否保留顶层的全连接网络
- weights：None代表随机初始化，即不加载预训练权重。'imagenet'代表加载预训练权重
- input_tensor：可填入Keras tensor作为模型的图像输出tensor
- pooling：当include_top=False时，该参数指定了池化方式。None代表不池化，最后一个卷积层的输出为4D张量。‘avg’代表全局平均池化，‘max’代表全局最大值池化。
- classes：可选，图片分类的类别数，仅当 `include_top=True` 并且不加载预训练权重时可用。

返回值

Keras 模型对象

参考文献

- Rethinking the Inception Architecture for Computer Vision：如果在研究中使用了InceptionV3，请引用该文

License

预训练权重由我们自己训练而来，基于MIT License

常用数据库

CIFAR10 小图片分类数据集

该数据库具有50,000个32*32的彩色图片作为训练集，10,000个图片作为测试集。图片一共有10个类别。

使用方法

```
from keras.datasets import cifar10

(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

返回值：

两个Tuple

`X_train` 和 `X_test` 是形如（nb_samples, 3, 32, 32）的RGB三通道图像数据，数据类型是无符号8位整形（uint8）

`Y_train` 和 `Y_test` 是形如（nb_samples,）标签数据，标签的范围是0~9

CIFAR100 小图片分类数据库

该数据库具有50,000个32*32的彩色图片作为训练集，10,000个图片作为测试集。图片一共有100个类别，每个类别有600张图片。这100个类别又分为20个大类。

使用方法

```
from keras.datasets import cifar100

(X_train, y_train), (X_test, y_test) = cifar100.load_data(label_
mode='fine')
```

参数

- label_mode：为‘fine’或‘coarse’之一，控制标签的精细度，‘fine’获得的标签是100个小类的标签，‘coarse’获得的标签是大类的标签

返回值

两个Tuple, `(X_train, y_train), (X_test, y_test)` ，其中

- X_train和X_test：是形如（nb_samples, 3, 32, 32）的RGB三通道图像数据，数据类型是无符号8位整形（uint8）
- y_train和y_test：是形如（nb_samples,）标签数据，标签的范围是0~9

IMDB影评倾向分类

本数据库含有来自IMDB的25,000条影评，被标记为正面/负面两种评价。影评已被预处理为词下标构成的序列。方便起见，单词的下标基于它在数据集中出现的频率标定，例如整数3所编码的词为数据集中第3常出现的词。这样的组织方法使得用户可以快速完成诸如“只考虑最常出现的10,000个词，但不考虑最常出现的20个词”这样的操作

按照惯例，0不代表任何特定的词，而用来编码任何未知单词

使用方法

```
from keras.datasets import imdb

(X_train, y_train), (X_test, y_test) = imdb.load_data(path="imdb
.npz",
                                                      nb_words=N
one,
                                                      skip_top=0
,
                                                      maxlen=None
,
                                                      test_split=
0.1)
                                                      seed=113,
                                                      start_char=
1,
                                                      oov_char=2
,
                                                      index_from=
3)
```

参数

- path：如果你在本机上已有此数据集（位于 `'~/.keras/datasets/'+path` ），则载入。否则数据将下载到该目录下
- nb_words：整数或None，要考虑的最常见的单词数，序列中任何出现频率更低的单词将会被编码为 `oov_char` 的值。
- skip_top：整数，忽略最常出现的若干单词，这些单词将会被编码为 `oov_char` 的值
- maxlen：整数，最大序列长度，任何长度大于此值的序列将被截断
- seed：整数，用于数据重排的随机数种子
- start_char：字符，序列的起始将以该字符标记，默认为1因为0通常用作padding

- oov_char：整数，因 `nb_words` 或 `skip_top` 限制而cut掉的单词将被该字符代替

- index_from：整数，真实的单词（而不是类似于 `start_char` 的特殊占位符）将从这个下标开始

返回值

两个Tuple, `(X_train, y_train), (X_test, y_test)` ，其中

- X_train和X_test：序列的列表，每个序列都是词下标的列表。如果指定了 `nb_words` ，则序列中可能的最大下标为 `nb_words-1` 。如果指定了 `maxlen` ，则序列的最大可能长度为 `maxlen`

- y_train和y_test：为序列的标签，是一个二值list

路透社新闻主题分类

本数据库包含来自路透社的11,228条新闻，分为了46个主题。与IMDB库一样，每条新闻被编码为一个词下标的序列。

使用方法

```
from keras.datasets import reuters


(X_train, y_train), (X_test, y_test) = reuters.load_data(path="r
euters.npz",
                                                         nb_word
s=None,
                                                         skip_to
p=0,
                                                         maxlen=
None,
                                                         test_sp
lit=0.2,
                                                         seed=113
,
                                                         start_c
har=1,
                                                         oov_cha
r=2,
                                                         index_f
rom=3)
```

参数的含义与IMDB同名参数相同，唯一多的参数是：

```
```python
word_index = reuters.get_word_index(path="reuters_word_index.jso
n")
```

上面代码的返回值是一个以单词为关键字，以其下标为值的字典。例如，`word_index['giraffe']` 的值可能为 `1234`

参数

- path：如果你在本机上已有此数据集（位于 `'~/.keras/datasets/'+path` ），则载入。否则数据将下载到该目录下
```

MNIST手写数字识别

本数据库有60,000个用于训练的28*28的灰度手写数字图片，10,000个测试图片

使用方法

```
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

参数

- path：如果你在本机上已有此数据集（位于 `'~/.keras/datasets/'+path` ），则载入。否则数据将下载到该目录下

返回值

两个Tuple, `(X_train, y_train), (X_test, y_test)` ，其中

- X_train和X_test：是形如（nb_samples, 28, 28）的灰度图片数据，数据类型是无符号8位整形（uint8）

- y_train和y_test：是形如（nb_samples,）标签数据，标签的范围是0~9

数据库将会被下载到 `'~/.keras/datasets/'+path`

Boston房屋价格回归数据集

本数据集由StatLib库取得，由CMU维护。每个样本都是1970s晚期波士顿郊区的不同位置，每条数据含有13个属性，目标值是该位置房子的房价中位数（千dollar）。

使用方法

```
from keras.datasets import boston_housing

(x_train, y_train), (x_test, y_test) = boston_housing.load_data(
)
```

参数

- path：数据存放位置，默认 `'~/.keras/datasets/'+path`
- seed：随机数种子
- test_split：分割测试集的比例

返回值

两个Tuple, `(X_train, y_train), (X_test, y_test)`

模型可视化

```
该函数将画出模型结构图，并保存成图片：

```python
from keras.utils import plot_model
plot_model(model, to_file='model.png')
```

```
* ```show_shapes```：指定是否显示输出数据的形状，默认为```False
```

- `show_layer_names` :指定是否显示层名称,默认为 `True`

我们也可以直接获取一个 `pydot.Graph` 对象，然后按照自己的需要配置它，例如，如果要在ipython中展示图片

```python
from IPython.display import SVG
from keras.utils.visualize_util import model_to_dot

SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

【Tips】依赖 pydot-ng 和 graphviz，若出现错误，用命令行输入 `pip install pydot-ng & brew install graphviz`
```

utils 工具

本模块提供了一系列有用工具

CustomObjectScope

```
keras.utils.generic_utils.CustomObjectScope()
```

提供定制类的作用域，在该作用域内全局定制类能够被更改，但在作用域结束后将回到初始状态。 以 `with` 声明开头的代码将能够通过名字访问定制类的实例，在with的作用范围，这些定制类的变动将一直持续，在with作用域结束后，全局定制类的实例将回归其在with作用域前的状态。

```
with CustomObjectScope({"MyObject":MyObject}):
    layer = Dense(..., W_regularizer="MyObject")
    # save, load, etc. will recognize custom object by name
```

HDF5Matrix

```
keras.utils.io_utils.HDF5Matrix(datapath, dataset, start=0, end=
None, normalizer=None)
```

这是一个使用HDF5数据集代替Numpy数组的方法

提供 `start` 和 `end` 参数可以进行切片，另外，还可以提供一个正规化函数或匿名函数，该函数将会在每片数据检索时自动调用。

```
x_data = HDF5Matrix('input/file.hdf5', 'data')
model.predict(x_data)
```

- datapath: 字符串，HDF5文件的路径
- dataset: 字符串，在datapath路径下HDF5数据库名字

- start: 整数，想要的数据切片起点
- end: 整数，想要的数据切片终点
- normalizer: 在每个切片数据检索时自动调用的函数对象

to_categorical

```
to_categorical(y, num_classes=None)
```

将类别向量(从0到nb_classes的整数向量)映射为二值类别矩阵, 用于应用到以 `categorical_crossentropy` 为目标函数的模型中.

参数

- y: 类别向量
- num_classes:总共类别数

normalize

```
normalize(x, axis=-1, order=2)
```

对numpy数组规范化，返回规范化后的数组

参数

- x：待规范化的数据
- axis: 规范化的轴
- order：规范化方法，如2为L2范数

convert_all_kernels_in_model

```
convert_all_kernels_in_model(model)
```

将模型中全部卷积核在Theano和TensorFlow模式中切换

plot_model

```
plot_model(model, to_file='model.png', show_shapes=False, show_layer_names=True)
```

绘制模型图

custom_object_scope

```
custom_object_scope()
```

提供定制类的作用域，在该作用域内全局定制类能够被更改，但在作用域结束后将回到初始状态。 以 `with` 声明开头的代码将能够通过名字访问定制类的实例，在with的作用范围，这些定制类的变动将一直持续，在with作用域结束后，全局定制类的实例将回归其在with作用域前的状态。

本函数返回 `CustomObjectScope` 对象

```
with custom_object_scope({"MyObject":MyObject}):
    layer = Dense(..., W_regularizer="MyObject")
    # save, load, etc. will recognize custom object by name
```

get_custom_objects

```
get_custom_objects()
```

检索全局定制类，推荐利用custom_object_scope更新和清理定制对象，但 `get_custom_objects` 可被直接用于访问 `_GLOBAL_CUSTOM_OBJECTS` 。本函数返回从名称到类别映射的全局字典。

```
get_custom_objects().clear()
get_custom_objects()["MyObject"] = MyObject
```

serialize_keras_object

```
serialize_keras_object(instance)
```

将keras对象序列化

deserialize_keras_object

```
eserialize_keras_object(identifier, module_objects=None, custom_
objects=None, printable_module_name='object')
```

从序列中恢复keras对象

get_file

```
get_file(fname, origin, untar=False, md5_hash=None, file_hash=No
ne, cache_subdir='datasets', hash_algorithm='auto', extract=False
, archive_format='auto', cache_dir=None)
```

从给定的URL中下载文件, 可以传递MD5值用于数据校验(下载后或已经缓存的数据均可)

默认情况下文件会被下载到 `~/.keras` 中的 `cache_subdir` 文件夹，并将其文件名设为 `fname` ，因此例如一个文件 `example.txt` 最终将会被存放在 `~/.keras/datasets/example.txt~

tar,tar.gz.tar.bz和zip格式的文件可以被提取，提供哈希码可以在下载后校验文件。命令喊程序 `shasum` 和 `sha256sum` 可以计算哈希值。

参数

- fname: 文件名，如果指定了绝对路径 `/path/to/file.txt` ,则文件将会保存到该位置。
- origin: 文件的URL地址
- untar: 布尔值,是否要进行解压
- md5_hash: MD5哈希值,用于数据校验，支持 `sha256` 和 `md5` 哈希
- cache_subdir: 用于缓存数据的文件夹，若指定绝对路径 `/path/to/folder` 则将存放在该路径下。
- hash_algorithm: 选择文件校验的哈希算法，可选项有'md5', 'sha256', 和'auto'. 默认'auto'自动检测使用的哈希算法
- extract: 若为True则试图提取文件，例如tar或zip tries extracting the file as an Archive, like tar or zip.
- archive_format: 试图提取的文件格式，可选为'auto', 'tar', 'zip', 和None. 'tar' 包括tar, tar.gz, tar.bz文件. 默认'auto'是['tar', 'zip']. None或空列表将返回没有匹配。
- cache_dir: 缓存文件存放地在，参考FAQ

返回值

下载后的文件地址

Keras后端

什么是“后端”

Keras是一个模型级的库，提供了快速构建深度学习网络的模块。Keras并不处理如张量乘法、卷积等底层操作。这些操作依赖于某种特定的、优化良好的张量操作库。Keras依赖于处理张量的库就称为“后端引擎”。Keras提供了两种后端引擎Theano/Tensorflow，并将其函数统一封装，使得用户可以以同一个接口调用不同后端引擎的函数

- Theano是一个开源的符号主义张量操作框架，由蒙特利尔大学LISA/MILA实验室开发
- TensorFlow是一个符号主义的张量操作框架，由Google开发

在未来，我们有可能要添加更多的后端选项。下一步我们很可能加入的是CNTK后端。

切换后端

注意：Windows用户请把 `$Home` 换为 `%USERPROFILE%`

如果你至少运行过一次Keras，你将在下面的目录下找到Keras的配置文件：

```
如果该目录下没有该文件，你可以手动创建一个

文件的默认配置如下：
```

{ "image_data_format": "channels_last", "epsilon": 1e-07, "floatx": "float32", "backend": "tensorflow" }

```
将```backend```字段的值改写为你需要使用的后端：```theano```或```tensor
flow```，即可完成后端的切换

我们也可以通过定义环境变量```KERAS_BACKEND```来覆盖上面配置文件中定义的后
端：

```python
KERAS_BACKEND=tensorflow python -c "from keras import backend;"
Using TensorFlow backend.
```

keras.json 细节

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

你可以更改以上 `~/.keras/keras.json` 中的配置

- `iamge_data_format` ：字符串，"channels_last"或"channels_first"，该选项指定了Keras将要使用的维度顺序，可通过 `keras.backend.image_data_format()` 来获取当前的维度顺序。对2D数据来说，"channels_last"假定维度顺序为(rows,cols,channels)而"channels_first"假定维度顺序为(channels, rows, cols)。对3D数据而言，"channels_last"假定(conv_dim1, conv_dim2, conv_dim3, channels)，"channels_first"则是(channels, conv_dim1, conv_dim2, conv_dim3)
- `epsilon` ：浮点数，防止除0错误的小数字
- `floatx` ：字符串，"float16", "float32", "float64"之一，为浮点数精度
- `backend` ：字符串，所使用的后端，为"tensorflow"或"theano"
```

使用抽象的Keras后端来编写代码

如果你希望你编写的Keras模块能够同时在Theano和TensorFlow两个后端上使用，你可以通过Keras后端接口来编写代码，这里是一个简介：

```
from keras import backend as K
```

下面的代码实例化了一个输入占位符，等价于 `tf.placeholder()` ， `T.matrix()` ， `T.tensor3()` 等

```
input = K.placeholder(shape=(2, 4, 5))
# also works:
input = K.placeholder(shape=(None, 4, 5))
# also works:
input = K.placeholder(ndim=3)
```

下面的代码实例化了一个共享变量（shared），等价于 `tf.variable()` 或 `theano.shared()`

```
val = np.random.random((3, 4, 5))
var = K.variable(value=val)

# all-zeros variable:
var = K.zeros(shape=(3, 4, 5))
# all-ones:
var = K.ones(shape=(3, 4, 5))
```

大多数你需要的张量操作都可以通过统一的Keras后端接口完成，而不关心具体执行这些操作的是Theano还是TensorFlow

```
a = b + c * K.abs(d)
c = K.dot(a, K.transpose(b))
a = K.sum(b, axis=2)
a = K.softmax(b)
a = concatenate([b, c], axis=-1)
# etc...
```

Kera后端函数

backend

```
backend()
```

返回当前后端

epsilon

```
epsilon()
```

以数值形式返回一个（一般来说很小的）数，用以防止除0错误

set_epsilon

```
set_epsilon(e)
```

设置在数值表达式中使用的fuzz factor，用于防止除0错误，该值应该是一个较小的浮点数，示例：

```
>>> from keras import backend as K
>>> K.epsilon()
1e-08
>>> K.set_epsilon(1e-05)
>>> K.epsilon()
1e-05
```

floatx

```
floatx()
```

返回默认的浮点数数据类型，为字符串，如 'float16', 'float32', 'float64'

set_floatx(floatx)

```
floatx()
```

设置默认的浮点数数据类型，为字符串，如 'float16', 'float32', 'float64',示例：

```
>>> from keras import backend as K
>>> K.floatx()
'float32'
>>> K.set_floatx('float16')
>>> K.floatx()
'float16'
```

cast_to_floatx

```
cast_to_floatx(x)
```

将numpy array转换为默认的Keras floatx类型，x为numpy array，返回值也为numpy array但其数据类型变为floatx。示例：

```
>>> from keras import backend as K
>>> K.floatx()
'float32'
>>> arr = numpy.array([1.0, 2.0], dtype='float64')
>>> arr.dtype
dtype('float64')
>>> new_arr = K.cast_to_floatx(arr)
>>> new_arr
array([ 1.,  2.], dtype=float32)
>>> new_arr.dtype
dtype('float32')
```

image_data_format

```
image_data_format()
```

返回默认的图像的维度顺序（'channels_last'或'channels_first'）

set_image_data_format

```
set_image_data_format(data_format)
```

设置图像的维度顺序（'tf'或'th'）,示例：

> > > from keras import backend as K K.image_data_format() 'channels_first' K.set_image_data_format('channels_last') K.image_data_format() 'channels_last' ```

is_keras_tensor()

```
is_keras_tensor(x)
```

判断x是否是keras tensor对象的谓词函数

```
>>> from keras import backend as K
>>> np_var = numpy.array([1, 2])
>>> K.is_keras_tensor(np_var)
False
>>> keras_var = K.variable(np_var)
>>> K.is_keras_tensor(keras_var)  # A variable is not a Tensor.
False
>>> keras_placeholder = K.placeholder(shape=(2, 4, 5))
>>> K.is_keras_tensor(keras_placeholder)  # A placeholder is a T
ensor.
True
```

get_uid

```
get_uid(prefix='')
```

获得默认计算图的uid，依据给定的前缀提供一个唯一的UID，参数为表示前缀的字符串，返回值为整数.

reset_uids

```
reset_uids()
```

重置图的标识符

is_keras_tensor

```
is_keras_tensor(x)
```

判断x是否是一个Keras tensor，返回一个布尔值，示例

```
>>> from keras import backend as K
>>> np_var = numpy.array([1, 2])
>>> K.is_keras_tensor(np_var)
False
>>> keras_var = K.variable(np_var)
>>> K.is_keras_tensor(keras_var)  # A variable is not a Tensor.
False
>>> keras_placeholder = K.placeholder(shape=(2, 4, 5))
>>> K.is_keras_tensor(keras_placeholder)  # A placeholder is a T
ensor.
True
```

clear_session

```
clear_session()
```

结束当前的TF计算图，并新建一个。有效的避免模型/层的混乱

manual_variable_initialization

```
manual_variable_initialization(value)
```

指出变量应该以其默认值被初始化还是由用户手动初始化，参数value为布尔值，默认False代表变量由其默认值初始化

learning_phase

```
learning_phase()
```

返回训练模式/测试模式的flag，该flag是一个用以传入Keras模型的标记，以决定当前模型执行于训练模式下还是测试模式下

set_learning_phase

```
set_learning_phase()
```

设置训练模式/测试模式0或1

is_sparse

```
is_sparse(tensor)
```

判断一个tensor是不是一个稀疏的tensor(稀不稀疏由tensor的类型决定，而不是tensor实际上有多稀疏)，返回值是一个布尔值，示例：

```
>>> from keras import backend as K
>>> a = K.placeholder((2, 2), sparse=False)
>>> print(K.is_sparse(a))
False
>>> b = K.placeholder((2, 2), sparse=True)
>>> print(K.is_sparse(b))
True
```

to_dense

```
to_dense(tensor)
```

将一个稀疏tensor转换一个不稀疏的tensor并返回之，示例：

```
>>> from keras import backend as K
>>> b = K.placeholder((2, 2), sparse=True)
>>> print(K.is_sparse(b))
True
>>> c = K.to_dense(b)
>>> print(K.is_sparse(c))
False
```

variable

```
variable(value, dtype='float32', name=None)
```

实例化一个张量，返回之

参数：

- value：用来初始化张量的值
- dtype：张量数据类型
- name：张量的名字（可选）

示例：

```
>>> from keras import backend as K
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val, dtype='float64', name='example_
var')
>>> K.dtype(kvar)
'float64'
>>> print(kvar)
example_var
>>> kvar.eval()
array([[ 1.,  2.],
   [ 3.,  4.]])
```

placeholder

```
placeholder(shape=None, ndim=None, dtype='float32', name=None)
```

实例化一个占位符，返回之

参数：

- shape：占位符的shape（整数tuple，可能包含None）
- ndim: 占位符张量的阶数，要初始化一个占位符，至少指定 `shape` 和 `ndim` 之一，如果都指定则使用 `shape`
- dtype: 占位符数据类型
- name: 占位符名称（可选）

示例：

```
>>> from keras import backend as K
>>> input_ph = K.placeholder(shape=(2, 4, 5))
>>> input_ph._keras_shape
(2, 4, 5)
>>> input_ph
<tf.Tensor 'Placeholder_4:0' shape=(2, 4, 5) dtype=float32>
```

shape

```
shape(x)
```

返回一个张量的符号shape，符号shape的意思是返回值本身也是一个tensor，示例：

```
>>> from keras import backend as K
>>> tf_session = K.get_session()
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> input = keras.backend.placeholder(shape=(2, 4, 5))
>>> K.shape(kvar)
<tf.Tensor 'Shape_8:0' shape=(2,) dtype=int32>
>>> K.shape(input)
<tf.Tensor 'Shape_9:0' shape=(3,) dtype=int32>
__To get integer shape (Instead, you can use K.int_shape(x))__

>>> K.shape(kvar).eval(session=tf_session)
array([2, 2], dtype=int32)
>>> K.shape(input).eval(session=tf_session)
array([2, 4, 5], dtype=int32)
```

int_shape

```
int_shape(x)
```

以整数Tuple或None的形式返回张量shape，示例：

```
>>> from keras import backend as K
>>> input = K.placeholder(shape=(2, 4, 5))
>>> K.int_shape(input)
(2, 4, 5)
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> K.int_shape(kvar)
(2, 2)
```

ndim

```
ndim(x)
```

返回张量的阶数，为整数，示例：

```
>>> from keras import backend as K
>>> input = K.placeholder(shape=(2, 4, 5))
>>> val = np.array([[1, 2], [3, 4]])
>>> kvar = K.variable(value=val)
>>> K.ndim(input)
3
>>> K.ndim(kvar)
2
```

dtype

```
dtype(x)
```

返回张量的数据类型，为字符串，示例：

```
>>> from keras import backend as K
>>> K.dtype(K.placeholder(shape=(2,4,5)))
'float32'
>>> K.dtype(K.placeholder(shape=(2,4,5), dtype='float32'))
'float32'
>>> K.dtype(K.placeholder(shape=(2,4,5), dtype='float64'))
'float64'
__Keras variable__

>>> kvar = K.variable(np.array([[1, 2], [3, 4]]))
>>> K.dtype(kvar)
'float32_ref'
>>> kvar = K.variable(np.array([[1, 2], [3, 4]]), dtype='float32'
)
>>> K.dtype(kvar)
'float32_ref'
```

eval

```
eval(x)
```

求得张量的值，返回一个Numpy array，示例：

```
>>> from keras import backend as K
>>> kvar = K.variable(np.array([[1, 2], [3, 4]]), dtype='float32'
)
>>> K.eval(kvar)
array([[ 1.,  2.],
   [ 3.,  4.]], dtype=float32)
```

zeros

```
zeros(shape, dtype='float32', name=None)
```

生成一个全0张量，示例：

```
>>> from keras import backend as K
>>> kvar = K.zeros((3,4))
>>> K.eval(kvar)
array([[ 0.,  0.,  0.,  0.],
   [ 0.,  0.,  0.,  0.],
   [ 0.,  0.,  0.,  0.]], dtype=float32)
```

ones

```
ones(shape, dtype='float32', name=None)
```

生成一个全1张量，示例

```
>>> from keras import backend as K
>>> kvar = K.ones((3,4))
>>> K.eval(kvar)
array([[ 1.,  1.,  1.,  1.],
   [ 1.,  1.,  1.,  1.],
   [ 1.,  1.,  1.,  1.]], dtype=float32)
```

eye

```
eye(size, dtype='float32', name=None)
```

生成一个单位矩阵，示例：

```
>>> from keras import backend as K
>>> kvar = K.eye(3)
>>> K.eval(kvar)
array([[ 1.,  0.,  0.],
   [ 0.,  1.,  0.],
   [ 0.,  0.,  1.]], dtype=float32)
```

zeros_like

```
zeros_like(x, name=None)
```

生成与另一个张量x的shape相同的全0张量，示例：

```
>>> from keras import backend as K
>>> kvar = K.variable(np.random.random((2,3)))
>>> kvar_zeros = K.zeros_like(kvar)
>>> K.eval(kvar_zeros)
array([[ 0.,  0.,  0.],
   [ 0.,  0.,  0.]], dtype=float32)
```

ones_like

```
ones_like(x, name=None)
```

生成与另一个张量shape相同的全1张量，示例：

```
>>> from keras import backend as K
>>> kvar = K.variable(np.random.random((2,3)))
>>> kvar_ones = K.ones_like(kvar)
>>> K.eval(kvar_ones)
array([[ 1.,  1.,  1.],
   [ 1.,  1.,  1.]], dtype=float32)
```

random_uniform_variable

```
random_uniform_variable(shape, low, high, dtype=None, name=None,
 seed=None)
```

初始化一个Keras变量，其数值为从一个均匀分布中采样的样本，返回之。

参数：

- shape：张量shape
- low：浮点数，均匀分布之下界
- high：浮点数，均匀分布之上界
- dtype：数据类型
- name：张量名
- seed：随机数种子

示例：

```
>>> kvar = K.random_uniform_variable((2,3), 0, 1)
>>> kvar
<tensorflow.python.ops.variables.Variable object at 0x10ab40b10>
>>> K.eval(kvar)
array([[ 0.10940075,  0.10047495,  0.476143  ],
   [ 0.66137183,  0.00869417,  0.89220798]], dtype=float32)
```

count_params

```
count_params(x)
```

返回张量中标量的个数，示例：

```
>>> kvar = K.zeros((2,3))
>>> K.count_params(kvar)
6
>>> K.eval(kvar)
array([[ 0.,  0.,  0.],
   [ 0.,  0.,  0.]], dtype=float32)
```

cast

```
cast(x, dtype)
```

改变张量的数据类型，dtype只能是 `float16` ， `float32` 或 `float64` 之一，示例：

```
>>> from keras import backend as K
>>> input = K.placeholder((2, 3), dtype='float32')
>>> input
<tf.Tensor 'Placeholder_2:0' shape=(2, 3) dtype=float32>
__It doesn't work in-place as below.__

>>> K.cast(input, dtype='float16')
<tf.Tensor 'Cast_1:0' shape=(2, 3) dtype=float16>
>>> input
<tf.Tensor 'Placeholder_2:0' shape=(2, 3) dtype=float32>
__you need to assign it.__

>>> input = K.cast(input, dtype='float16')
>>> input
<tf.Tensor 'Cast_2:0' shape=(2, 3) dtype=float16>
```

```


### update
```python
update(x, new_x)
```

用new_x更新x

update_add

```
update_add(x, increment)
```

通过将x增加increment更新x

update_sub

```
update_sub(x, decrement)
```

通过将x减少decrement更新x
```

moving_average_update

```
moving_average_update(x, value, momentum)
```

含义暂不明确

dot

```
dot(x, y)
```

求两个张量的乘积。当试图计算两个N阶张量的乘积时，与Theano行为相同，如 `(2, 3).(4, 3, 5) = (2, 4, 5))` ，示例：

```
>>> x = K.placeholder(shape=(2, 3))
>>> y = K.placeholder(shape=(3, 4))
>>> xy = K.dot(x, y)
>>> xy
<tf.Tensor 'MatMul_9:0' shape=(2, 4) dtype=float32>
```

```
>>> x = K.placeholder(shape=(32, 28, 3))
>>> y = K.placeholder(shape=(3, 4))
>>> xy = K.dot(x, y)
>>> xy
<tf.Tensor 'MatMul_9:0' shape=(32, 28, 4) dtype=float32>
```

Theano-like的行为示例：

```
>>> x = K.random_uniform_variable(shape=(2, 3), low=0, high=1)
>>> y = K.ones((4, 3, 5))
>>> xy = K.dot(x, y)
>>> K.int_shape(xy)
(2, 4, 5)
```

batch_dot

```
batch_dot(x, y, axes=None)
```

按批进行张量乘法，该函数用于计算x和y的点积，其中x和y都是成batch出现的数据。即它的数据shape形如 `(batch_size,:)` 。batch_dot将产生比输入张量维度低的张量，如果张量的维度被减至1，则通过 `expand_dims` 保证其维度至少为2 例如，假设 `x = [[1, 2],[3,4]]` ， `y = [[5, 6],[7, 8]]` ，则 `batch_dot(x, y, axes=1) = [[17, 53]]` ，即 `x.dot(y.T)` 的主对角元素，此过程中我们没有计算过反对角元素的值

参数：

- x,y：阶数大于等于2的张量，在tensorflow下，只支持大于等于3阶的张量
- axes：目标结果的维度，为整数或整数列表， `axes[0]` 和 `axes[1]` 应相同

示例： 假设 `x=[[1,2],[3,4]]` ， `y=[[5,6],[7,8]]` ，则 `batch_dot(x, y, axes=1)` 为 `[[17, 53]]` ，恰好为 `x.dot(y.T)` 的主对角元，整个过程没有计算反对角元的元素。

我们做一下shape的推导，假设x是一个shape为(100,20)的tensor，y是一个shape为(100,30,20)的tensor，假设 `axes=(1,2)` ，则输出tensor的shape通过循环x.shape和y.shape确定：

- `x.shape[0]` ：值为100，加入到输入shape里
- `x.shape[1]` ：20，不加入输出shape里，因为该维度的值会被求和(dot_axes[0]=1)
- `y.shape[0]` ：值为100，不加入到输出shape里，y的第一维总是被忽略
- `y.shape[1]` ：30，加入到输出shape里
- `y.shape[2]` ：20，不加到output shape里，y的第二个维度会被求和(dot_axes[1]=2)

- 结果为(100, 30)

```
>>> x_batch = K.ones(shape=(32, 20, 1))
>>> y_batch = K.ones(shape=(32, 30, 20))
>>> xy_batch_dot = K.batch_dot(x_batch, y_batch, axes=[1, 2])
>>> K.int_shape(xy_batch_dot)
(32, 1, 30)
```

transpose

```
transpose(x)
```

张量转置，返回转置后的tensor，示例：

```
>>> var = K.variable([[1, 2, 3], [4, 5, 6]])
>>> K.eval(var)
array([[ 1.,  2.,  3.],
   [ 4.,  5.,  6.]], dtype=float32)
>>> var_transposed = K.transpose(var)
>>> K.eval(var_transposed)
array([[ 1.,  4.],
   [ 2.,  5.],
   [ 3.,  6.]], dtype=float32)

>>> input = K.placeholder((2, 3))
>>> input
<tf.Tensor 'Placeholder_11:0' shape=(2, 3) dtype=float32>
>>> input_transposed = K.transpose(input)
>>> input_transposed
<tf.Tensor 'transpose_4:0' shape=(3, 2) dtype=float32>
```

gather

```
gather(reference, indices)
```

在给定的张量中检索给定下标的向量

参数：

- reference：张量
- indices：整数张量，其元素为要查询的下标

返回值：一个与 `reference` 数据类型相同的张量

max

```
max(x, axis=None, keepdims=False)
```

求张量中的最大值

min

```
min(x, axis=None, keepdims=False)
```

求张量中的最小值

sum

```
sum(x, axis=None, keepdims=False)
```

在给定轴上计算张量中元素之和

prod

```
prod(x, axis=None, keepdims=False)
```

在给定轴上计算张量中元素之积

cumsum

```
cumsum(x, axis=0)
```

在给定轴上求张量的累积和

cumprod

```
cumprod(x, axis=0)
```

在给定轴上求张量的累积积

var

```
var(x, axis=None, keepdims=False)
```

在给定轴上计算张量方差

std

```
std(x, axis=None, keepdims=False)
```

在给定轴上求张量元素之标准差

mean

```
mean(x, axis=None, keepdims=False)
```

在给定轴上求张量元素之均值

any

```
any(x, axis=None, keepdims=False)
```

按位或，返回数据类型为uint8的张量（元素为0或1）

all

```
any(x, axis=None, keepdims=False)
```

按位与，返回类型为uint8de tensor

argmax

```
argmax(x, axis=-1)
```

在给定轴上求张量之最大元素下标

argmin

```
argmin(x, axis=-1)
```

在给定轴上求张量之最小元素下标

square

```
square(x)
```

逐元素平方

abs

```
abs(x)
```

逐元素绝对值

sqrt

```
sqrt(x)
```

逐元素开方

exp

```
exp(x)
```

逐元素求自然指数

log

```
log(x)
```

逐元素求自然对数

logsumexp

```
logsumexp(x, axis=None, keepdims=False)
```

在给定轴上计算log(sum(exp()))，该函数在数值稳定性上超过直接计算log(sum(exp()))，可以避免由exp和log导致的上溢和下溢

round

```
round(x)
```

逐元素四舍五入

sign

```
sign(x)
```

逐元素求元素的符号（+1或-1）

pow

```
pow(x, a)
```

逐元素求x的a次方

clip

```
clip(x, min_value, max_value)
```

逐元素clip（将超出指定范围的数强制变为边界值）

equal

```
equal(x, y)
```

逐元素判相等关系，返回布尔张量

not_equal

```
not_equal(x, y)
```

逐元素判不等关系，返回布尔张量

greater

```
greater(x,y)
```

逐元素判断x>y关系，返回布尔张量

greater_equal

```
greater_equal(x,y)
```

逐元素判断x>=y关系，返回布尔张量

lesser

```
lesser(x,y)
```

逐元素判断x<y关系，返回布尔张量

lesser_equal

```
lesser_equal(x,y)
```

逐元素判断x<=y关系，返回布尔张量

maximum

```
maximum(x, y)
```

逐元素取两个张量的最大值

minimum

```
minimum(x, y)
```

逐元素取两个张量的最小值

sin

```
sin(x)
```

逐元素求正弦值

cos

```
cos(x)
```

逐元素求余弦值

normalize_batch_in_training

```
normalize_batch_in_training(x, gamma, beta, reduction_axes, epsilon=0.0001)
```

对一个batch数据先计算其均值和方差，然后再进行batch_normalization

batch_normalization

```
batch_normalization(x, mean, var, beta, gamma, epsilon=0.0001)
```

对一个batch的数据进行batch_normalization，计算公式为： output = (x-mean)/(sqrt(var)+epsilon)*gamma+beta

concatenate

```
concatenate(tensors, axis=-1)
```

在给定轴上将一个列表中的张量串联为一个张量 specified axis

reshape

```
reshape(x, shape)
```

将张量的shape变换为指定shape

permute_dimensions

```
permute_dimensions(x, pattern)
```

按照给定的模式重排一个张量的轴

参数：

- pattern：代表维度下标的tuple如 `(0, 2, 1)`

resize_images

```
resize_images(X, height_factor, width_factor, dim_ordering)
```

依据给定的缩放因子，改变一个batch图片的shape，参数中的两个因子都为正整数，图片的排列顺序与维度的模式相关，如'th'和'tf'

resize_volumes

```
resize_volumes(X, depth_factor, height_factor, width_factor, dim_ordering)
```

依据给定的缩放因子，改变一个5D张量数据的shape，参数中的两个因子都为正整数，图片的排列顺序与维度的模式相关，如'th'和'tf'。5D数据的形式是batch, channels, depth, height, width或batch, depth, height, width, channels

repeat_elements

```
repeat_elements(x, rep, axis)
```

在给定轴上重复张量元素 `rep` 次，与 `np.repeat` 类似。例如，若xshape `(s1, s2, s3)` 并且给定轴为 `axis=1`，输出张量的shape为`(s1, s2 * rep, s3)`

repeat

```
repeat(x, n)
```

重复2D张量，例如若xshape是 `(samples, dim)` 且n为2，则输出张量的shape是 `(samples, 2, dim)`

arange

```
arange(start, stop=None, step=1, dtype='int32')
```

生成1D的整数序列张量，该函数的参数与Theano的arange函数含义相同，如果只有一个参数被提供了，那么它实际上就是 `stop` 参数的值

为了与tensorflow的默认保持匹配，函数返回张量的默认数据类型是 `int32`

tile

```
tile(x, n)
```

将x在各个维度上重复n次，x为张量，n为与x维度数目相同的列表

batch_flatten

```
batch_flatten(x)
```

将一个n阶张量转变为2阶张量，其第一维度保留不变

expand_dims

```
expand_dims(x, dim=-1)
```

在下标为 `dim` 的轴上增加一维

squeeze

```
squeeze(x, axis)
```

将下标为 `axis` 的一维从张量中移除

temporal_padding

```
temporal_padding(x, padding=1)
```

向3D张量中间的那个维度的左右两端填充 `padding` 个0值

asymmetric_temporal_padding

```
asymmetric_temporal_padding(x, left_pad=1, right_pad=1)
```

向3D张量中间的那个维度的一端填充 `padding` 个0值

spatial_2d_padding

```
spatial_2d_padding(x, padding=(1, 1), dim_ordering='th')
```

向4D张量第二和第三维度的左右两端填充 `padding[0]` 和 `padding[1]` 个0值

spatial_3d_padding

```
spatial_3d_padding(x, padding=(1, 1, 1), dim_ordering='th')
```

向5D张量深度、高度和宽度三个维度上填充 `padding[0]` ， `padding[1]` 和 `padding[2]` 个0值

stack

```
stack(x, axis=0)
```

将一个列表中维度数目为R的张量堆积起来形成维度为R+1的新张量

one-hot

```
one_hot(indices, nb_classes)
```

输入为n维的整数张量，形如(batch_size, dim1, dim2, ... dim(n-1))，输出为(n+1)维的one-hot编码，形如(batch_size, dim1, dim2, ... dim(n-1), nb_classes)

reverse

```
reverse(x, axes)
```

将一个张量在给定轴上反转

get_value

```
get_value(x)
```

以Numpy array的形式返回张量的值

batch_get_value

```
batch_get_value(x)
```

以Numpy array list的形式返回多个张量的值

set_value

```
set_value(x, value)
```

从numpy array将值载入张量中

batch_set_value

```
batch_set_value(tuples)
```

将多个值载入多个张量变量中

参数：

- tuples: 列表，其中的元素形如 `(tensor, value)` 。 `value` 是要载入的Numpy array数据

print_tensor

```
print_tensor(x, message='')
```

在求值时打印张量的信息，并返回原张量

function

```
function(inputs, outputs, updates=[])
```

实例化一个Keras函数

参数：

- inputs:：列表，其元素为占位符或张量变量
- outputs：输出张量的列表
- updates：列表，其元素是形如 `(old_tensor, new_tensor)` 的tuple.

gradients

```
gradients(loss, variables)
```

返回loss函数关于variables的梯度，variables为张量变量的列表

stop_gradient

```
stop_gradient(variables)
```

Returns `variables` but with zero gradient with respect to every other variables.

rnn

```
rnn(step_function, inputs, initial_states, go_backwards=False, mask=None, constants=None, unroll=False, input_length=None)
```

在张量的时间维上迭代

参数：

- inputs： 形如 `(samples, time, ...)` 的时域信号的张量，阶数至少为3
- step_function：每个时间步要执行的函数 其参数：
  - input：形如 `(samples, ...)` 的张量，不含时间维，代表某个时间步时一个batch的样本
  - states：张量列表 其返回值：
  - output：形如 `(samples, ...)` 的张量
  - new_states：张量列表，与'states'的长度相同
- initial_states：形如 `(samples, ...)` 的张量，包含了 `step_function` 状态的初始值。
- go_backwards：布尔值，若设为True，则逆向迭代序列
- mask：形如 `(samples, time, 1)` 的二值张量，需要屏蔽的数据元素上值为1
- constants：按时间步传递给函数的常数列表
- unroll：当使用TensorFlow时，RNN总是展开的。当使用Theano时，设置该值为 `True` 将展开递归网络

- input_length：使用TensorFlow时不需要此值，在使用Theano时，如果要展开递归网络，必须指定输入序列

返回值：形如 `(last_output, outputs, new_states)` 的tuple

- last_output：rnn最后的输出，形如 `(samples, ...)`
- outputs：形如 `(samples, time, ...)` 的张量，每个在[s,t]点的输出对应于样本s在t时间的输出
- new_states: 列表，其元素为形如 `(samples, ...)` 的张量，代表每个样本的最后一个状态

switch

```
switch(condition, then_expression, else_expression)
```

依据给定的条件'condition'（整数或布尔值）在两个表达式之间切换，注意两个表达式都应该是具有同样shape的符号化张量表达式

参数：

- condition：标量张量
- then_expression：TensorFlow表达式
- else_expression: TensorFlow表达式

in_train_phase

```
in_train_phase(x, alt)
```

如果处于训练模式，则选择x，否则选择alt，注意alt应该与x的shape相同

in_test_phase

```
in_test_phase(x, alt)
```

如果处于测试模式，则选择x，否则选择alt，注意alt应该与x的shape相同

relu

```
relu(x, alpha=0.0, max_value=None)
```

修正线性单元

参数：

- alpha：负半区斜率
- max_value: 饱和门限

elu

```
elu(x, alpha=1.0)
```

指数线性单元

参数：

- x：输入张量
- alpha: 标量

softmax

```
softmax(x)
```

返回张量的softmax值

softplus

```
softplus(x)
```

返回张量的softplus值

softsign

```
softsign(x)
```

返回张量的softsign值

categorical_crossentropy

```
categorical_crossentropy(output, target, from_logits=False)
```

计算输出张量和目标张量的Categorical crossentropy（类别交叉熵），目标张量与输出张量必须shape相同

sparse_categorical_crossentropy

```
sparse_categorical_crossentropy(output, target, from_logits=False
)
```

计算输出张量和目标张量的Categorical crossentropy（类别交叉熵），目标张量必须是整型张量

binary_crossentropy

```
binary_crossentropy(output, target, from_logits=False)
```

计算输出张量和目标张量的交叉熵

sigmoid

```
sigmoid(x)
```

逐元素计算sigmoid值

hard_sigmoid

```
hard_sigmoid(x)
```

该函数是分段线性近似的sigmoid，计算速度更快

tanh

```
tanh(x)
```

逐元素计算sigmoid值

dropout

```
dropout(x, level, seed=None)
```

随机将x中一定比例的值设置为0，并放缩整个tensor

参数：

- x：张量
- level：x中设置成0的元素比例
- seed：随机数种子

l2_normalize

```
l2_normalize(x, axis)
```

在给定轴上对张量进行L2范数规范化

in_top_k

```
in_top_k(predictions, targets, k)
```

判断目标是否在predictions的前k大值位置

参数：

- predictions：预测值张量, shape为(batch_size, classes), 数据类型float32
- targets：真值张量, shape为(batch_size,),数据类型为int32或int64
- k：整数

conv1d

```
conv1d(x, kernel, strides=1, border_mode='valid', image_shape=None, filter_shape=None)
```

1D卷积

参数：

- kernel：卷积核张量
- strides：步长，整型
- border_mode：“same”，“valid”之一的字符串

conv2d

```
conv2d(x, kernel, strides=(1, 1), border_mode='valid', dim_ordering='th', image_shape=None, filter_shape=None)
```

2D卷积

参数：

- kernel：卷积核张量
- strides：步长，长为2的tuple
- border_mode：“same”，“valid”之一的字符串
- dim_ordering：“tf”和“th”之一，维度排列顺序

deconv2d

```
deconv2d(x, kernel, output_shape, strides=(1, 1), border_mode='valid', dim_ordering='th', image_shape=None, filter_shape=None)
```

2D反卷积（转置卷积）

参数：

- x：输入张量
- kernel：卷积核张量
- output_shape: 输出shape的1D的整数张量
- strides：步长，tuple类型
- border_mode：“same”或“valid”
- dim_ordering：“tf”或“th”

conv3d

```
conv3d(x, kernel, strides=(1, 1, 1), border_mode='valid', dim_ordering='th', volume_shape=None, filter_shape=None)
```

3D卷积

参数：

- x：输入张量
- kernel：卷积核张量
- strides：步长，tuple类型
- border_mode：“same”或“valid”
- dim_ordering：“tf”或“th”

pool2d

```
pool2d(x, pool_size, strides=(1, 1), border_mode='valid', dim_ordering='th', pool_mode='max')
```

2D池化

参数：

- pool_size：含有两个整数的tuple，池的大小
- strides：含有两个整数的tuple，步长
- border_mode：“same”，“valid”之一的字符串
- dim_ordering：“tf”和“th”之一，维度排列顺序
- pool_mode: “max”，“avg”之一，池化方式

pool3d

```
pool3d(x, pool_size, strides=(1, 1, 1), border_mode='valid', dim_ordering='th', pool_mode='max')
```

3D池化

参数：

- pool_size：含有3个整数的tuple，池的大小
- strides：含有3个整数的tuple，步长
- border_mode：“same”，“valid”之一的字符串
- dim_ordering：“tf”和“th”之一，维度排列顺序
- pool_mode: “max”，“avg”之一，池化方式

bias_add

```
bias_add(x, bias, data_format=None)
```

为张量增加一个偏置项

random_normal

```
random_normal(shape, mean=0.0, stddev=1.0, dtype=None, seed=None)
```

返回具有正态分布值的张量，mean和stddev为均值和标准差

random_uniform

```
random_uniform(shape, minval=0.0, maxval=1.0, dtype=None, seed=None)
```

返回具有均匀分布值的张量，minval和maxval是均匀分布的下上界

random_binomial

```
random_binomial(shape, p=0.0, dtype=None, seed=None)
```

返回具有二项分布值的张量，p是二项分布参数

truncated_normall

```
truncated_normal(shape, mean=0.0, stddev=1.0, dtype=None, seed=None)
```

返回具有截尾正态分布值的张量，在距离均值两个标准差之外的数据将会被截断并重新生成

ctc_label_dense_to_sparse

```
ctc_label_dense_to_sparse(labels, label_lengths)
```

将ctc标签从稠密形式转换为稀疏形式

ctc_batch_cost

```
ctc_batch_cost(y_true, y_pred, input_length, label_length)
```

在batch上运行CTC损失算法

参数：

- y_true：形如(samples，max_tring_length)的张量，包含标签的真值
- y_pred：形如(samples，time_steps，num_categories)的张量，包含预测值或输出的softmax值
- input_length：形如(samples，1)的张量，包含y_pred中每个batch的序列长
- label_length：形如(samples，1)的张量，包含y_true中每个batch的序列长

返回值：形如(samoles，1)的tensor，包含了每个元素的CTC损失

ctc_decode

```
ctc_decode(y_pred, input_length, greedy=True, beam_width=None, dict_seq_lens=None, dict_values=None)
```

使用贪婪算法或带约束的字典搜索算法解码softmax的输出

参数：

- y_pred：形如(samples，time_steps，num_categories)的张量，包含预测值或输出的softmax值
- input_length：形如(samples，1)的张量，包含y_pred中每个batch的序列长
- greedy：设置为True使用贪婪算法，速度快
- dict_seq_lens：dic_values列表中各元素的长度
- dict_values：列表的列表，代表字典

返回值：形如(samples，time_steps，num_catgories)的张量，包含了路径可能性（以softmax概率的形式）。注意仍然需要一个用来取出argmax和处理空白标签的函数

map_fn

```
map_fn(fn, elems, name=None)
```

元素elems在函数fn上的映射，并返回结果

参数：

- fn：函数
- elems：张量
- name：节点的名字

返回值：返回一个张量，该张量的第一维度等于elems，第二维度取决于fn

foldl

```
foldl(fn, elems, initializer=None, name=None)
```

减少elems，用fn从左到右连接它们

参数：

- fn：函数，例如：lambda acc, x: acc + x
- elems：张量
- initializer：初始化的值(elems[0])
- name：节点名

返回值：与initializer的类型和形状一致

foldr

```
foldr(fn, elems, initializer=None, name=None)
```

减少elems，用fn从右到左连接它们

参数：

- fn：函数，例如：lambda acc, x: acc + x
- elems：张量
- initializer：初始化的值（elems[-1]）
- name：节点名

返回值：与initializer的类型和形状一致

Scikit-Learn接口包装器

我们可以通过包装器将 `Sequential` 模型（仅有一个输入）作为Scikit-Learn工作流的一部分，相关的包装器定义在 `keras.wrappers.scikit_learn.py` 中

目前，有两个包装器可用：

`keras.wrappers.scikit_learn.KerasClassifier(build_fn=None, **sk_params)` 实现了sklearn的分类器接口

`keras.wrappers.scikit_learn.KerasRegressor(build_fn=None, **sk_params)` 实现了sklearn的回归器接口

参数

- build_fn：可调用的函数或类对象
- sk_params：模型参数和训练参数

`build_fn` 应构造、编译并返回一个Keras模型，该模型将稍后用于训练/测试。 `build_fn` 的值可能为下列三种之一：

1. 一个函数
2. 一个具有 `call` 方法的类对象
3. None，代表你的类继承自 `KerasClassifier` 或 `KerasRegressor` ，其 `call` 方法为其父类的 `call` 方法

`sk_params` 以模型参数和训练（超）参数作为参数。合法的模型参数为 `build_fn` 的参数。注意，'build_fn'应提供其参数的默认值。所以我们不传递任何值给 `sk_params` 也可以创建一个分类器/回归器

`sk_params` 还接受用于调用 `fit` ， `predict` ， `predict_proba` 和 `score` 方法的参数，如 `nb_epoch` ， `batch_size` 等。这些用于训练或预测的参数按如下顺序选择：

1. 传递给 `fit` ， `predict` ， `predict_proba` 和 `score` 的字典参数
2. 传递个 `sk_params` 的参数

3. `keras.models.Sequential`，`fit`，`predict`，`predict_proba`和`score`的默认值

当使用scikit-learn的`grid_search`接口时，合法的可转换参数是你可以传递给`sk_params`的参数，包括训练参数。即，你可以使用`grid_search`来搜索最佳的`batch_size`或`nb_epoch`以及其他模型参数

深度学习与Keras

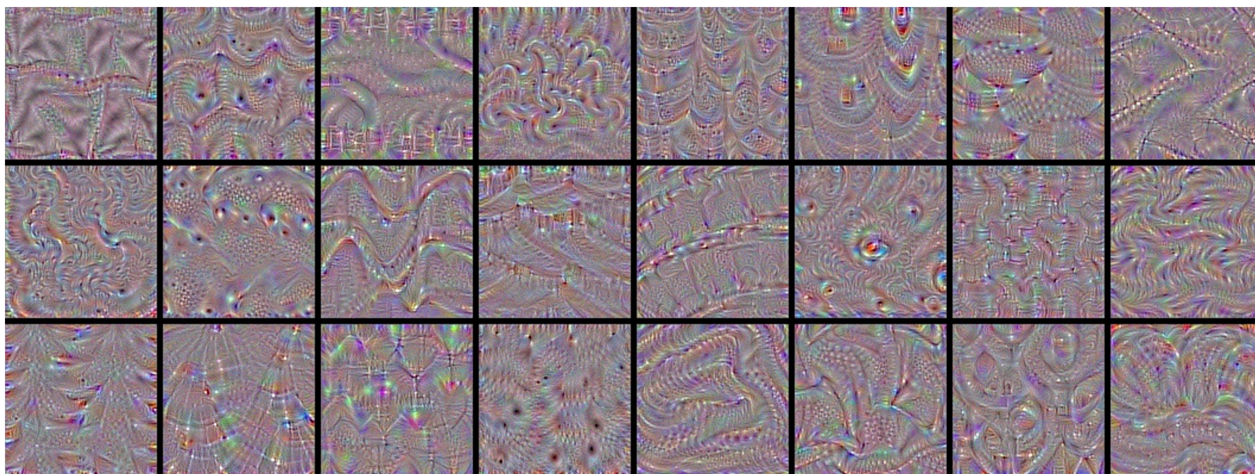
CNN眼中的世界：利用Keras解释CNN的滤波器

文章信息

本文地址：http://blog.keras.io/how-convolutional-neural-networks-see-the-world.html 本文作者：Francois Chollet

使用Keras探索卷积网络的滤波器

本文中我们将利用Keras观察CNN到底在学些什么，它是如何理解我们送入的训练图片的。我们将使用Keras来对滤波器的激活值进行可视化。本文使用的神经网络是VGG-16，数据集为ImageNet。本文的代码可以在[github](https://github.com/fchollet/keras/blob/master/examples/conv_filter_visualization.py)找到



VGG-16又称为OxfordNet，是由牛津视觉几何组（Visual Geometry Group）开发的卷积神经网络结构。该网络赢得了ILSVR（ImageNet）2014的冠军。时至今日，VGG仍然被认为是一个杰出的视觉模型——尽管它的性能实际上已经被后来的Inception和ResNet超过了。

Lorenzo Baraldi将Caffe预训练好的VGG16和VGG19模型转化为了Keras权重文件，所以我们可以简单的通过载入权重来进行实验。该权重文件可以在这里下载。国内的同学需要自备梯子。（这里是一个网盘保持的

vgg16：http://files.heuritech.com/weights/vgg16_weights.h5赶紧下载，网盘什么的不知道什么时候就挂了。）

首先，我们在Keras中定义VGG网络的结构：

```
from keras.models import Sequential
from keras.layers import Convolution2D, ZeroPadding2D, MaxPoolin
g2D

img_width, img_height = 128, 128

# build the VGG16 network
model = Sequential()
model.add(ZeroPadding2D((1, 1), batch_input_shape=(1, 3, img_wid
th, img_height)))
first_layer = model.layers[-1]
# this is a placeholder tensor that will contain our generated i
mages
input_img = first_layer.input

# build the rest of the network
model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1
_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1
_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu', name='conv
2_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu', name='conv
2_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv
3_1'))
model.add(ZeroPadding2D((1, 1)))
```

```
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

# get the symbolic outputs of each "key" layer (we gave them unique names).
layer_dict = dict([(layer.name, layer) for layer in model.layers])
```

注意我们不需要全连接层，所以网络就定义到最后一个卷积层为止。使用全连接层会将输入大小限制为224×224，即ImageNet原图片的大小。这是因为如果输入的图片大小不是224×224，在从卷积过度到全链接时向量的长度与模型指定的长度不相符。

下面，我们将预训练好的权重载入模型，一般而言我们可以通过 `model.load_weights()` 载入，但这里我们只载入一部分参数，如果使用该方法的话，模型和参数形式就不匹配了。所以我们需要手工载入：

```
import h5py

weights_path = 'vgg16_weights.h5'

f = h5py.File(weights_path)
for k in range(f.attrs['nb_layers']):
    if k >= len(model.layers):
        # we don't look at the last (fully-connected) layers in the savefile
        break
    g = f['layer_{}'.format(k)]
    weights = [g['param_{}'.format(p)] for p in range(g.attrs['nb_params'])]
    model.layers[k].set_weights(weights)
f.close()
print('Model loaded.')
```

下面，我们要定义一个损失函数，这个损失函数将用于最大化某个指定滤波器的激活值。以该函数为优化目标优化后，我们可以真正看一下使得这个滤波器激活的究竟是些什么东西。

现在我们使用Keras的后端来完成这个损失函数，这样这份代码不用修改就可以在TensorFlow和Theano之间切换了。TensorFlow在CPU上进行卷积要块的多，而目前为止Theano在GPU上进行卷积要快一些。

```
from keras import backend as K

layer_name = 'conv5_1'
filter_index = 0  # can be any integer from 0 to 511, as there a
re 512 filters in that layer

# build a loss function that maximizes the activation
# of the nth filter of the layer considered
layer_output = layer_dict[layer_name].output
loss = K.mean(layer_output[:, filter_index, :, :])

# compute the gradient of the input picture wrt this loss
grads = K.gradients(loss, input_img)[0]

# normalization trick: we normalize the gradient
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

# this function returns the loss and grads given the input pictu
re
iterate = K.function([input_img], [loss, grads])
```

注意这里有个小trick，计算出来的梯度进行了正规化，使得梯度不会过小或过大。这种正规化能够使梯度上升的过程平滑进行。

根据刚刚定义的函数，现在可以对某个滤波器的激活值进行梯度上升。

```
import numpy as np

# we start from a gray image with some noise
input_img_data = np.random.random((1, 3, img_width, img_height))
 * 20 + 128.
# run gradient ascent for 20 steps
for i in range(20):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step
```

使用TensorFlow时，这个操作大概只要几秒。

然后我们可以提取出结果，并可视化：

```
from scipy.misc import imsave

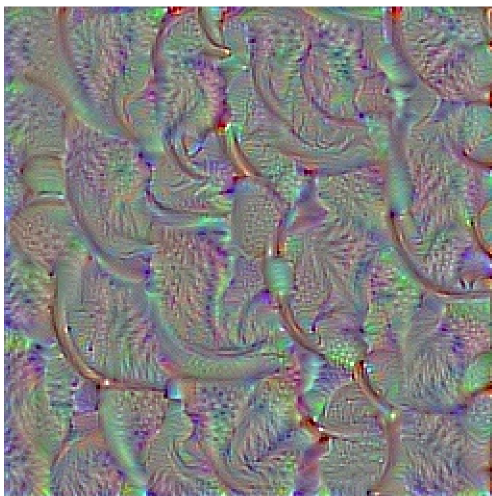
# util function to convert a tensor into a valid image
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = x.transpose((1, 2, 0))
    x = np.clip(x, 0, 255).astype('uint8')
    return x

img = input_img_data[0]
img = deprocess_image(img)
imsave('%s_filter_%d.png' % (layer_name, filter_index), img)
```

这里是第5卷基层第0个滤波器的结果：

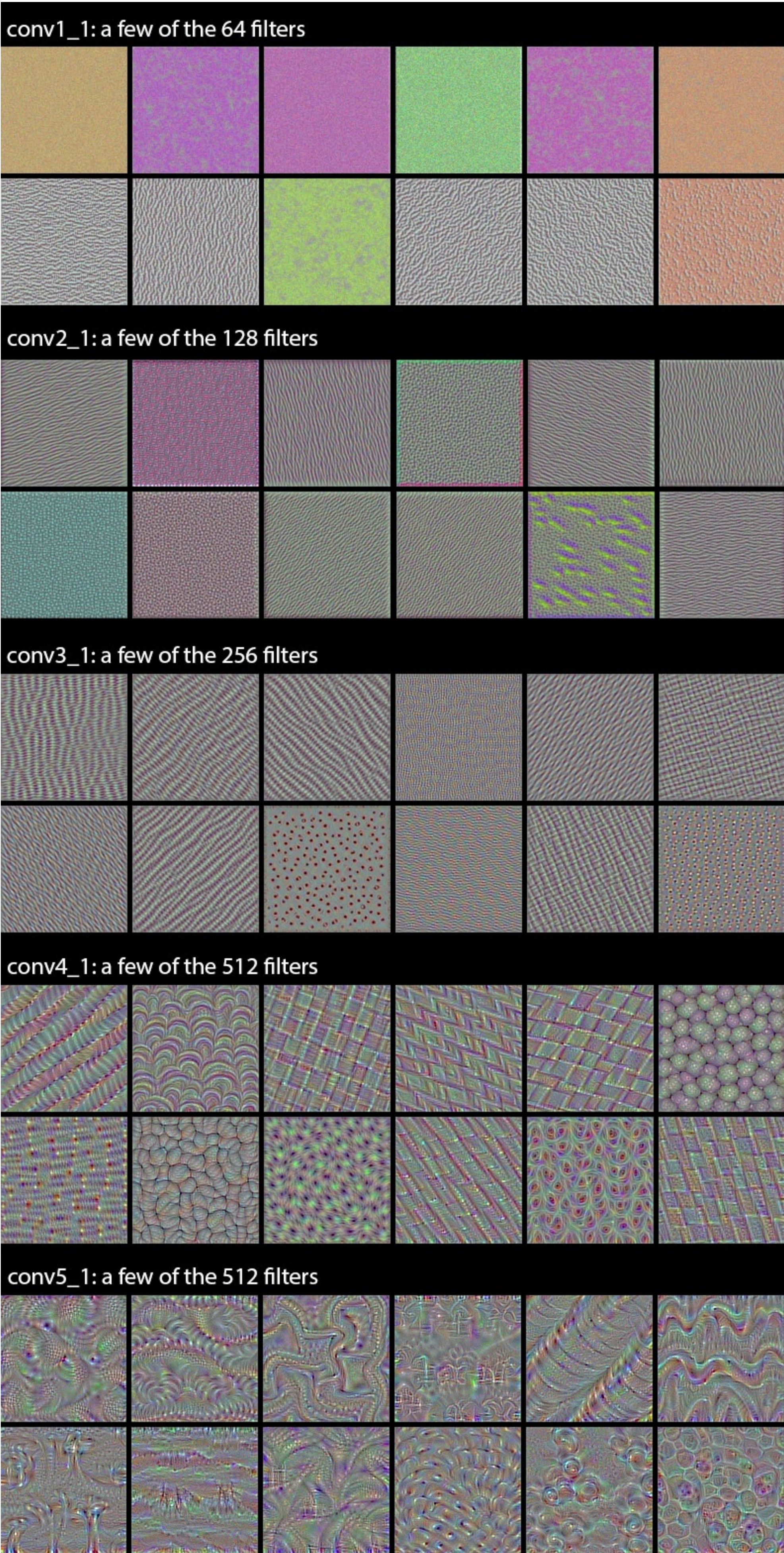


可视化所有的滤波器

下面我们系统的可视化一下各个层的各个滤波器结果，看看CNN是如何对输入进行逐层分解的。

第一层的滤波器主要完成方向、颜色的编码，这些颜色和方向与基本的纹理组合，逐渐生成复杂的形状。

可以将每层的滤波器想为基向量，这些基向量一般是过完备的。基向量可以将层的输入紧凑的编码出来。滤波器随着其利用的空域信息的拓宽而更加精细和复杂，

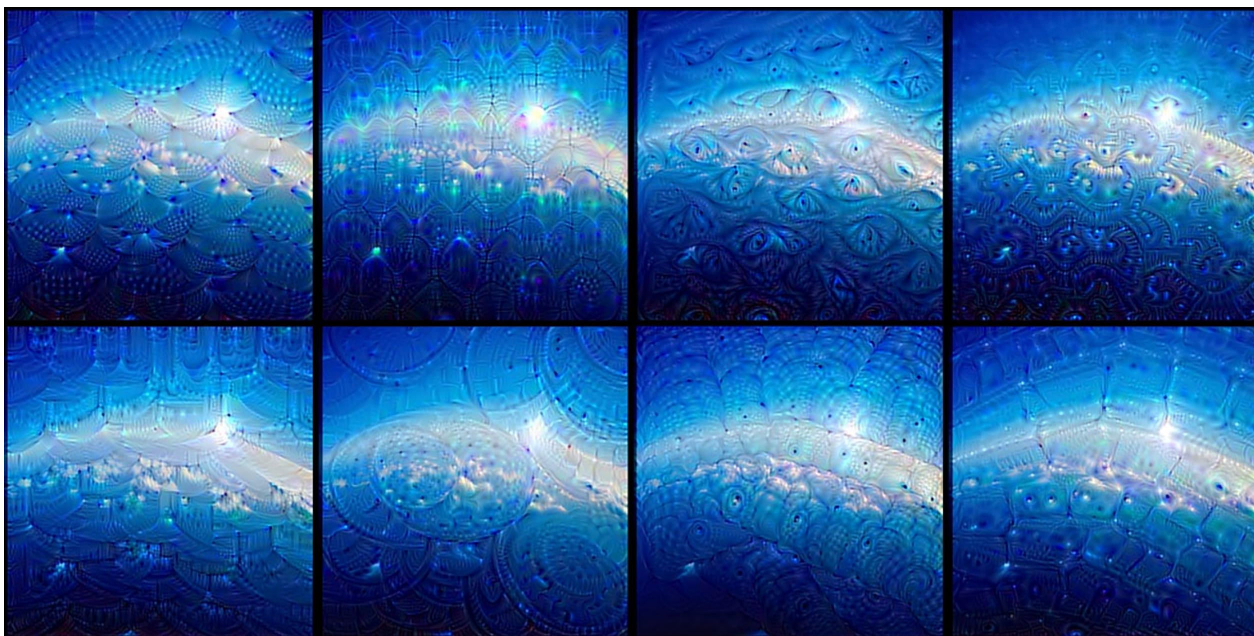

conv1_1: a few of the 64 filters
conv2_1: a few of the 128 filters
conv3_1: a few of the 256 filters
conv4_1: a few of the 512 filters
conv5_1: a few of the 512 filters

可以观察到，很多滤波器的内容其实是一样的，只不过旋转了一个随机的的角度（如90度）而已。这意味着我们可以通过使得卷积滤波器具有旋转不变性而显著减少滤波器的数目，这是一个有趣的研究方向。

令人震惊的是，这种旋转的性质在高层的滤波器中仍然可以被观察到。如Conv4_1

Deep Dream（nightmare）

另一个有趣的事儿是，如果我们把刚才的随机噪声图片替换为有意义的照片，结果就变的更好玩了。这就是去年由Google提出的Deep Dream。通过选择特定的滤波器组合，我们可以获得一些很有意思的结果。如果你对此感兴趣，可以参考Keras的例子 Deep Dream和Google的博客Google blog post（墙）



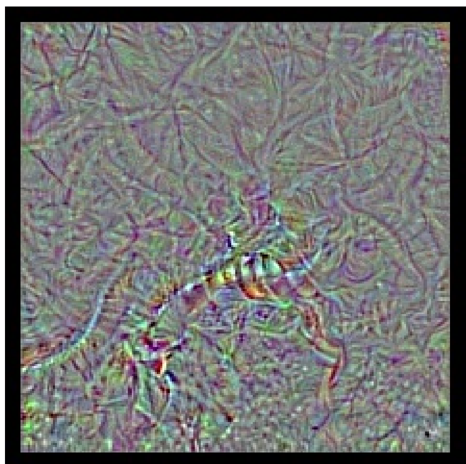
愚弄神经网络

如果我们添加上VGG的全连接层，然后试图最大化某个指定类别的激活值呢？你会得到一张很像该类别的图片吗？让我们试试。

这种情况下我们的损失函数长这样：

```
layer_output = model.layers[-1].get_output()
loss = K.mean(layer_output[:, output_index])
```

比方说我们来最大化输出下标为65的那个类，在ImageNet里，这个类是蛇。很快，我们的损失达到了0.999，即神经网络有99.9%的概率认为我们生成的图片是一条海蛇，它长这样：



不太像呀，换个类别试试，这次选喜鹊类（第18类）



"I am 99.99% certain this is a magpie", said the machine.

OK，我们的网络认为是喜鹊的东西看起来完全不是喜鹊，往好了说，这个图里跟喜鹊相似的，也不过就是一些局部的纹理，如羽毛，嘴巴之类的。那么，这就意味着卷积神经网络是个很差的工具吗？当然不是，我们按照一个特定任务来训练它，它就会在那个任务上表现的不错。但我们不能有网络"理解"某个概念的错觉。我们不能将网络人格化，它只是工具而已。比如一条狗，它能识别其为狗只是因为它能以很高的概率将其正确分类而已，而不代表它理解关于"狗"的任何外延。

革命尚未成功，同志仍需努力

所以，神经网络到底理解了什么呢？我认为有两件事是它们理解的。

其一，神经网络理解了如何将输入空间解耦为分层次的卷积滤波器组。其二，神经网络理解了从一系列滤波器的组合到一系列特定标签的概率映射。神经网络学习到的东西完全达不到人类的“看见”的意义，从科学的的角度讲，这当然也不意味着我们已经解决了计算机视觉的问题。想得别太多，我们才刚刚踩上计算机视觉天梯的第一步。

有些人说，卷积神经网络学习到的对输入空间的分层次解耦模拟了人类视觉皮层的行为。这种说法可能对也可能不对，但目前未知我们还没有比较强的证据来承认或否认它。当然，有些人可以期望人类的视觉皮层就是以类似的方式学东西的，某种程度上讲，这是对我们视觉世界的自然解耦（就像傅里叶变换是对周期声音信号的一种解耦一样自然）【译注：这里是说，就像声音信号的傅里叶变换表达了不同频率的声音信号这种很自然很物理的理解一样，我们可能会认为我们对视觉信息的识别就是分层来完成的，圆的是轮子，有四个轮子的是汽车，造型炫酷的汽车是跑车，像这样】。但是，人类对视觉信号的滤波、分层次、处理的本质很可能和我们弱鸡的卷积网络完全不是一回事。视觉皮层不是卷积的，尽管它们也分层，但那些层具有皮质列的结构，而这些结构的真正目的目前还不得而知，这种结构在我们的人工神经网络中还没有出现（尽管乔大帝Geoff Hinton正在在这个方面努力）。此外，人类有比给静态图像分类的感知器多得多的视觉感知器，这些感知器是连续而主动的，不是静态而被动的，这些感受器还被如眼动等多种机制复杂控制。

下次有风投或某知名CEO警告你要警惕我们深度学习的威胁时，想想上面说的吧。今天我们是有更好的工具来处理复杂的信息了，这很酷，但归根结底它们只是工具，而不是生物。它们做的任何工作在哪个宇宙的标准下都不够格称之为“思考”。在一个石头上画一个笑脸并不会使石头变得“开心”，尽管你的灵长目皮质会告诉你它很开心。

总而言之，卷积神经网络的可视化工作是很让人着迷的，谁能想到仅仅通过简单的梯度下降法和合理的损失函数，加上大规模的数据库，就能学到能很好解释复杂视觉信息的如此漂亮的分层模型呢。深度学习或许在实际的意义上并不智能，但它仍然能够达到几年前任何人都无法达到的效果。现在，如果我们能理解为什么深度学习如此有效，那……嘿嘿:)

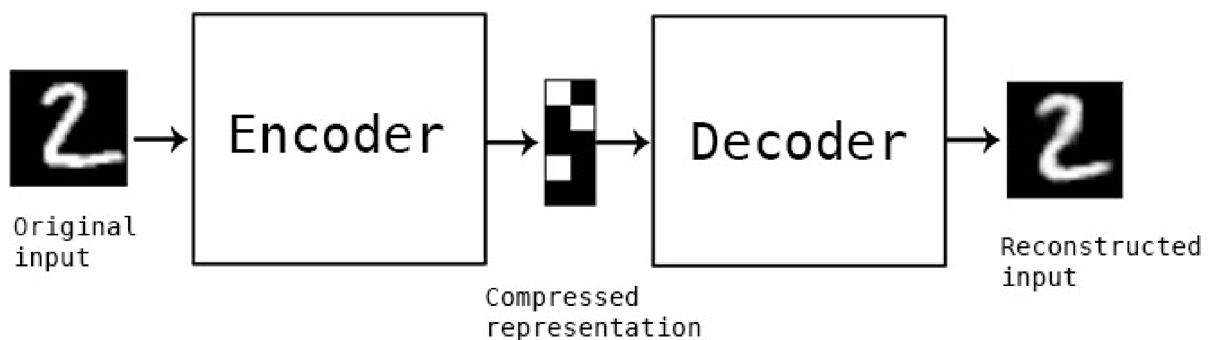
@fchollet, 2016年1月

自动编码器：各种各样的自动编码器

文章信息

本文地址：http://blog.keras.io/building-autoencoders-in-keras.html 本文作者：Francois Chollet

什么是自动编码器（Autoencoder）



自动编码器是一种数据的压缩算法，其中数据的压缩和解压缩函数是1）数据相关的,2）有损的，3）从样本中自动学习的。在大部分提到自动编码器的场合，压缩和解压缩的函数是通过神经网络实现的。

1）自动编码器是数据相关的（data-specific 或 data-dependent），这意味着自动编码器只能压缩那些与训练数据类似的数据。自编码器与一般的压缩算法，如MPEG-2，MP3等压缩算法不同，一般的通用算法只假设了数据是“图像”或“声音”，而没有指定是哪种图像或声音。比如，使用人脸训练出来的自动编码器在压缩别的图片，比如树木时性能很差，因为它学习到的特征是与人脸相关的。

2）自动编码器是有损的，意思是解压缩的输出与原来的输入相比是退化的，MP3，JPEG等压缩算法也是如此。这与无损压缩算法不同。

3）自动编码器是从数据样本中自动学习的，这意味着很容易对指定类的输入训练出一种特定的编码器，而不需要完成任何新工作。

搭建一个自动编码器需要完成下面三样工作：搭建编码器，搭建解码器，设定一个损失函数，用以衡量由于压缩而损失掉的信息。编码器和解码器一般都是参数化的方程，并关于损失函数可导，典型情况是使用神经网络。编码器和解码器的参数可以通过最小化损失函数而优化，例如SGD。

自编码器是一个好的数据压缩算法吗

通常情况下，使用自编码器做数据压缩，性能并不怎么样。以图片压缩为例，想要训练一个能和JPEG性能相提并论的自编码器非常困难，并且要达到这个性能，你还必须要把图片的类型限定在很小的一个范围内（例如JPEG不怎么行的某类图片）。自编码器依赖于数据的特性使得它在面对真实数据的压缩上并不可行，你只能在指定类型的数据上获得还可以的效果，但谁知道未来会有啥新需求？

那么，自编码器擅长做什么？

自编码器在实际应用中用的很少，2012年人们发现在卷积神经网络中使用自编码器做逐层预训练可以训练深度网络，但很快人们发现良好的初始化策略在训练深度网络上要比费劲的逐层预训练有效得多，2014年出现的Batch Normalization技术使得更深的网络也可以被有效训练，到了2015年底，通过使用残差学习（ResNet）我们基本上可以训练任意深度的神经网络。

目前自编码器的应用主要有两个方面，第一是数据去噪，第二是为进行可视化而降维。配合适当的维度和稀疏约束，自编码器可以学习到比PCA等技术更有意思的数据投影。

对于2D的数据可视化，t-SNE（读作tee-snee）或许是目前最好的算法，但通常还是需要原数据的维度相对低一些。所以，可视化高维数据的一个好办法是首先使用自编码器将维度降低到较低的水平（如32维），然后再使用t-SNE将其投影在2D平面上。Keras版本的t-SNE由Kyle McDonald实现了一下，放在了这里.ipynb)，另外scikit-learn也有一个简单实用的实现。

自编码器有什么卵用

自编码器的出名来自于网上很多机器学习课程的介绍，总而言之，一堆新手非常热爱自编码器而且怎么也玩不够，这就是这篇文章出现的意义【告诉你自编码器有什么卵用】。

自编码器吸引了一大批研究和关注的主要原因之一是很长时间一段以来它被认为是解决无监督学习的可能方案，即大家觉得自编码器可以在没有标签的时候学习到数据的有用表达。再说一次，自编码器并不是一个真正的无监督学习的算法，而是一个自监督的算法。自监督学习是监督学习的一个实例，其标签产生自输入数据。要获得一个自监督的模型，你需要想出一个靠谱的目标跟一个损失函数，问题来了，仅仅把目标设定为重构输入可能不是正确的选项。基本上，要求模型在像素级上精确重构输入不是机器学习的兴趣所在，学习到高级的抽象特征才是。事实上，当你的主要任务是分类、定位之类的任务时，那些对这类任务而言的最好的特征基本上都是重构输入时的最差的那种特征。

在应用自监督学习的视觉问题中，可能应用自编码器的领域有例如拼图，细节纹理匹配（从低分辨率的图像块中匹配其高分辨率的对应块）。下面这篇文章研究了拼图问题，其实很有意思，不妨一读。Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles.。此类问题的模型输入有些内置的假设，例如“视觉块比像素级的细节更重要”这样的，这种假设是普通的自编码器没有的。

Figure from Noroozi and Favaro (2016)



Fig. 1: What image representations do we learn by solving puzzles? Left: The image from which the tiles (marked with green lines) are extracted. Middle: A puzzle obtained by shuffling the tiles. Some tiles might be directly identifiable as object parts, but their identification is much more reliable once the correct ordering is found and the global figure emerges (Right).

使用Keras建立简单的自编码器

首先，先建立一个全连接的编码器和解码器

```
from keras.layers import Input, Dense
from keras.models import Model

# this is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, as
suming the input is 784 floats

# this is our input placeholder
input_img = Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input=input_img, output=decoded)
```

当然我们可以单独的使用编码器和解码器：

```
# this model maps an input to its encoded representation
encoder = Model(input=input_img, output=encoded)
```

```
# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(input=encoded_input, output=decoder_layer(encode
d_input))
```

下面我们训练自编码器，来重构MNIST中的数字，这里使用逐像素的交叉熵作为损失函数，优化器为adam

```
autoencoder.compile(optimizer='adadelta', loss='binary_crossentr
opy')
```

然后准备MNIST数据，将其归一化和向量化，然后训练：

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1
:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:]))
)
print x_train.shape
print x_test.shape

autoencoder.fit(x_train, x_train,
                nb_epoch=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

50个epoch后，看起来我们的自编码器优化的不错了，损失是0.10，我们可视化一下重构出来的输出：

```
# encode and decode some digits
# note that we take them from the *test* set
# use Matplotlib (don't ask)
import matplotlib.pyplot as plt

encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)


n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

这里是结果：



稀疏自编码器：为码字加上稀疏性约束

刚刚我们的隐层有32个神经元，这种情况下，一般而言自编码器学到的是PCA的一个近似（PCA不想科普了）。但是如果我们对隐层单元施加稀疏性约束的话，会得到更为紧凑的表达，只有一小部分神经元会被激活。在Keras中，我们可以通过添加一个activity_regularizer达到对某层激活值进行约束的目的：

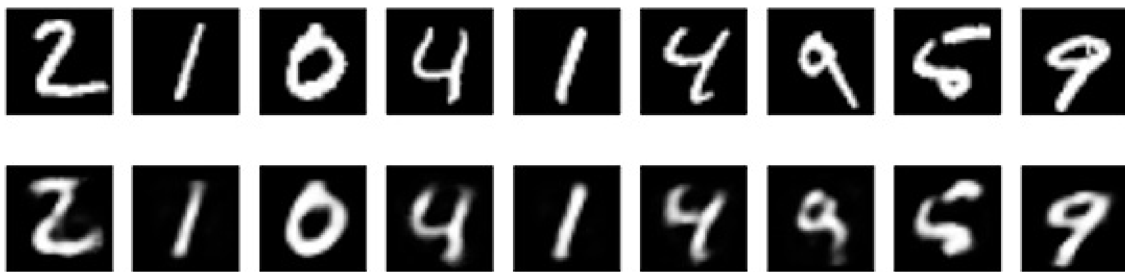
```
from keras import regularizers

encoding_dim = 32

input_img = Input(shape=(784,))
# add a Dense layer with a L1 activity regularizer
encoded = Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.activity_l1(10
e-5))(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input=input_img, output=decoded)
```

因为我们添加了正则性约束，所以模型过拟合的风险降低，我们可以训练多几次，这次训练100个epoch，得到损失为0.11，多出来的0.01基本上是由于正则项造成的。可视化结果如下：



结果上没有毛线差别，区别在于编码出来的码字更加稀疏了。稀疏自编码器的在10000个测试图片上的码字均值为3.33，而之前的为7.30

深度自编码器：把自编码器叠起来

把多个自编码器叠起来，像这样：

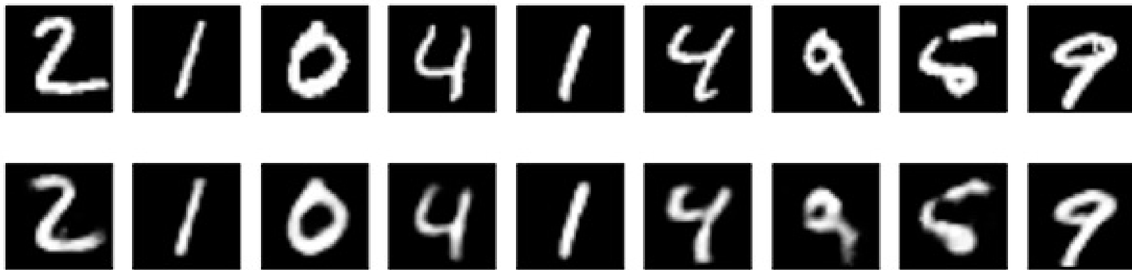
```
input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = Model(input=input_img, output=decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentr
opy')

autoencoder.fit(x_train, x_train,
                nb_epoch=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

100个epoch后，loss大概是0.097，比之前的模型好那么一丢丢



卷积自编码器：用卷积层搭建自编码器

当输入是图像时，使用卷积神经网络基本上总是有意义的。在现实中，用于处理图像的自动编码器几乎都是卷积自动编码器——又简单又快，棒棒哒

卷积自编码器的编码器部分由卷积层和MaxPooling层构成，MaxPooling负责空域下采样。而解码器由卷积层和上采样层构成。

```
from keras.layers import Input, Dense, Convolution2D, MaxPooling2D, UpSampling2D
from keras.models import Model

input_img = Input(shape=(1, 28, 28))

x = Convolution2D(16, 3, 3, activation='relu', border_mode='same')(input_img)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(x)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(x)
encoded = MaxPooling2D((2, 2), border_mode='same')(x)

# at this point the representation is (8, 4, 4) i.e. 128-dimensional

x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Convolution2D(8, 3, 3, activation='relu', border_mode='same')(x)
x = UpSampling2D((2, 2))(x)
x = Convolution2D(16, 3, 3, activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Convolution2D(1, 3, 3, activation='sigmoid', border_mode='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

我们使用28283的原始MNIST图像（尽管看起来还是灰度图）训练网络，图片的像素被归一化到0~1之间。

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 1, 28, 28))
x_test = np.reshape(x_test, (len(x_test), 1, 28, 28))
```

为了可视化训练过程的损失情况，我们使用TensorFlow作为后端，这样就可以启用TensorBoard了。打开一个终端并启动TensorBoard，TensorBoard将读取位于/tmp/autoencoder的日志文件：

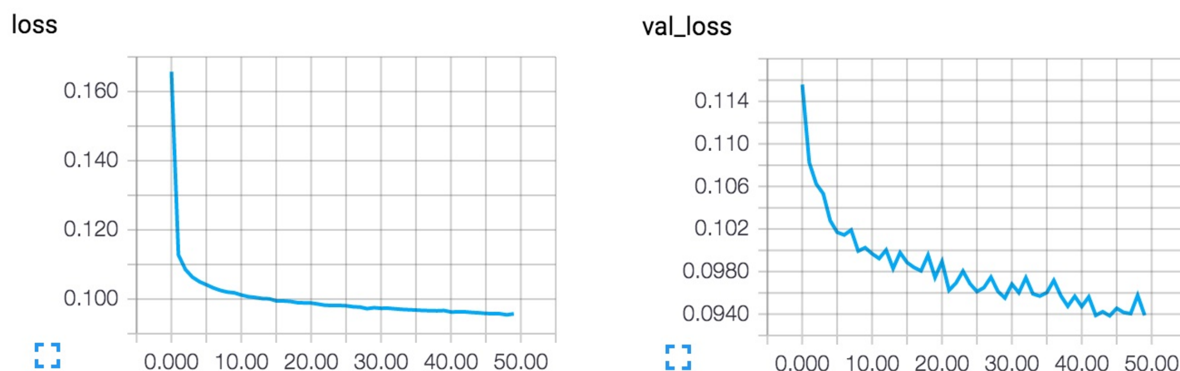
```
tensorboard --logdir=/tmp/autoencoder
```

然后我们把模型训练50个epoch，并在回调函数列表中传入TensorBoard回调函数，在每个epoch后回调函数将把训练的信息写入刚才的那个日志文件里，并被TensorBoard读取到

```
from keras.callbacks import TensorBoard

autoencoder.fit(x_train, x_train,
                nb_epoch=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/autoencoder'
)])
```

打开浏览器进入http://0.0.0.0:6006观测结果：

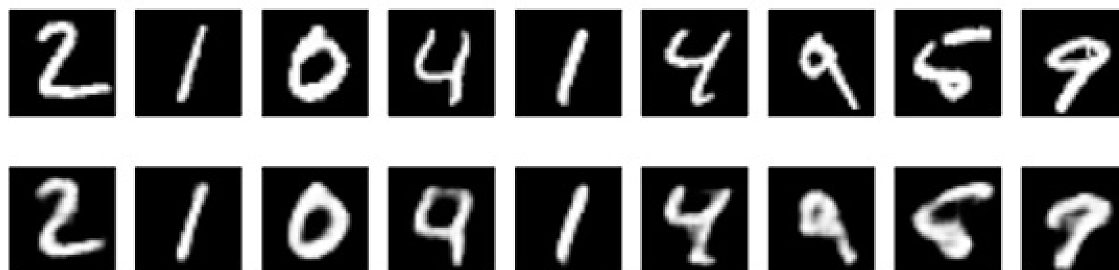


模型最后的loss是0.094，要比之前的模型都要好得多，因为现在我们的编码器的表达表达能力更强了。

```
decoded_imgs = autoencoder.predict(x_test)

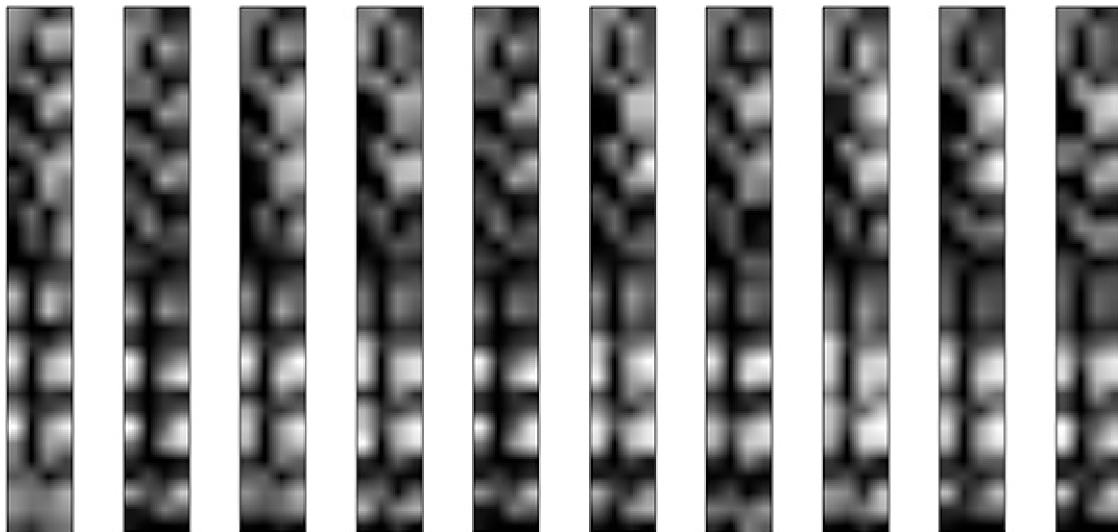
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



我们也可以看看中间的码字长什么样，这些码字的shape是844，我们可以将其reshape成4*32看

```
n = 10
plt.figure(figsize=(20, 8))
for i in range(n):
    ax = plt.subplot(1, n, i)
    plt.imshow(encoded_imgs[i].reshape(4, 4 * 8).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



使用自动编码器进行图像去噪

我们把训练样本用噪声污染，然后使解码器解码出干净的照片，以获得去噪自动编码器。首先我们把原图片加入高斯噪声，然后把像素值clip到0~1

```
from keras.datasets import mnist
import numpy as np

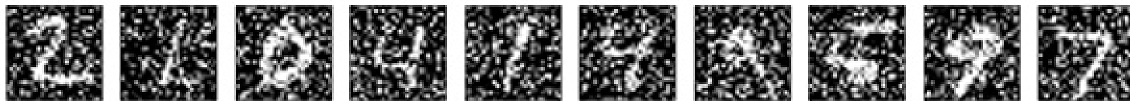
(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 1, 28, 28))
x_test = np.reshape(x_test, (len(x_test), 1, 28, 28))

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0
, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0,
 scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

我们可以先看看被污染的照片长啥样：



和之前的卷积自动编码器相比，为了提高重构图质量，我们的模型稍有不同

```
input_img = Input(shape=(1, 28, 28))

x = Convolution2D(32, 3, 3, activation='relu', border_mode='same'
)(input_img)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Convolution2D(32, 3, 3, activation='relu', border_mode='same'
)(x)
encoded = MaxPooling2D((2, 2), border_mode='same')(x)

# at this point the representation is (32, 7, 7)

x = Convolution2D(32, 3, 3, activation='relu', border_mode='same'
)(encoded)
x = UpSampling2D((2, 2))(x)
x = Convolution2D(32, 3, 3, activation='relu', border_mode='same'
)(x)
x = UpSampling2D((2, 2))(x)
decoded = Convolution2D(1, 3, 3, activation='sigmoid', border_mo
de='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentr
opy')
```

先来100个epoch的训练看看结果

```
autoencoder.fit(x_train_noisy, x_train,
                nb_epoch=100,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test_noisy, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/tb', histog
ram_freq=0, write_graph=False)])
```

结果如下，棒棒哒~



如果你将这个过程扩展到更大的卷积网络，你可以处理文档和声音的去噪，Kaggle有一个或许你会感兴趣的数据集在这里

序列到序列的自动编码器

如果你的输入是序列而不是2D的图像，那么你可能想要使用针对序列的模型构造自编码器，如LSTM。要构造基于LSTM的自编码器，首先我们需要一个LSTM的编码器来将输入序列变为一个向量，然后将这个向量重复N此，然后用LSTM的解码器将这个N步的时间序列变为目标序列。

这里我们不针对任何特定的数据库做这件事，只提供代码供读者参考

```
from keras.layers import Input, LSTM, RepeatVector
from keras.models import Model

inputs = Input(shape=(timesteps, input_dim))
encoded = LSTM(latent_dim)(inputs)

decoded = RepeatVector(timesteps)(encoded)
decoded = LSTM(input, return_sequences=True)(decoded)

sequence_autoencoder = Model(inputs, decoded)
encoder = Model(inputs, encoded)
```

变分自编码器（Variational autoencoder，VAE）：编码数据的分布

编码自编码器是更现代和有趣的一种自动编码器，它为码字施加约束，使得编码器学习到输入数据的隐变量模型。隐变量模型是连接显变量集和隐变量集的统计模型，隐变量模型的假设是显变量是由隐变量的状态控制的，各个显变量之间条件独

立。也就是说，变分编码器不再学习一个任意的函数，而是学习你的数据概率分布的一组参数。通过在这个概率分布中采样，你可以生成新的输入数据，即变分编码器是一个生成模型。

下面是变分编码器的工作原理：

首先，编码器网络将输入样本x转换为隐空间的两个参数，记作z_mean和z_log_sigma。然后，我们随机从隐藏的正态分布中采样得到数据点z，这个隐藏分布我们假设就是产生输入数据的那个分布。z = z_mean + exp(z_log_sigma)*epsilon，epsilon是一个服从正态分布的张量。最后，使用解码器网络将隐空间映射到显空间，即将z转换回原来的输入数据空间。

参数藉由两个损失函数来训练，一个是重构损失函数，该函数要求解码出来的样本与输入的样本相似（与之前的自编码器相同），第二项损失函数是学习到的隐分布与先验分布的KL距离，作为一个正则。实际上把后面这项损失函数去掉也可以，尽管它对学习符合要求的隐空间和防止过拟合有帮助。

因为VAE是一个很复杂的例子，我们把VAE的代码放在了github上，在这里。在这里我们来一步步回顾一下这个模型是如何搭建的

首先，建立编码网络，将输入影射为隐分布的参数：

```
x = Input(batch_shape=(batch_size, original_dim))
h = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim)(h)
z_log_sigma = Dense(latent_dim)(h)
```

然后从这些参数确定的分布中采样，这个样本相当于之前的隐层值

```
def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(batch_size, latent_dim),
                              mean=0., std=epsilon_std)
    return z_mean + K.exp(z_log_sigma) * epsilon

# note that "output_shape" isn't necessary with the TensorFlow b
ackend
# so you could write `Lambda(sampling)([z_mean, z_log_sigma])`
z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_
sigma])
```

最后，将采样得到的点映射回去重构原输入：

```
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)
```

到目前为止我们做的工作需要实例化三个模型：

- 一个端到端的自动编码器，用于完成输入信号的重构
- 一个用于将输入空间映射为隐空间的编码器
- 一个利用隐空间的分布产生的样本点生成对应的重构样本的生成器

```
# end-to-end autoencoder
vae = Model(x, x_decoded_mean)

# encoder, from inputs to latent space
encoder = Model(x, z_mean)

# generator, from latent space to reconstructed inputs
decoder_input = Input(shape=(latent_dim,))
_h_decoded = decoder_h(decoder_input)
_x_decoded_mean = decoder_mean(_h_decoded)
generator = Model(decoder_input, _x_decoded_mean)
```

我们使用端到端的模型训练，损失函数是一项重构误差，和一项KL距离

```
def vae_loss(x, x_decoded_mean):
    xent_loss = objectives.binary_crossentropy(x, x_decoded_mean
)
    kl_loss = - 0.5 * K.mean(1 + z_log_sigma - K.square(z_mean)
- K.exp(z_log_sigma), axis=-1)
    return xent_loss + kl_loss

vae.compile(optimizer='rmsprop', loss=vae_loss)
```

现在使用MNIST库来训练变分编码器：

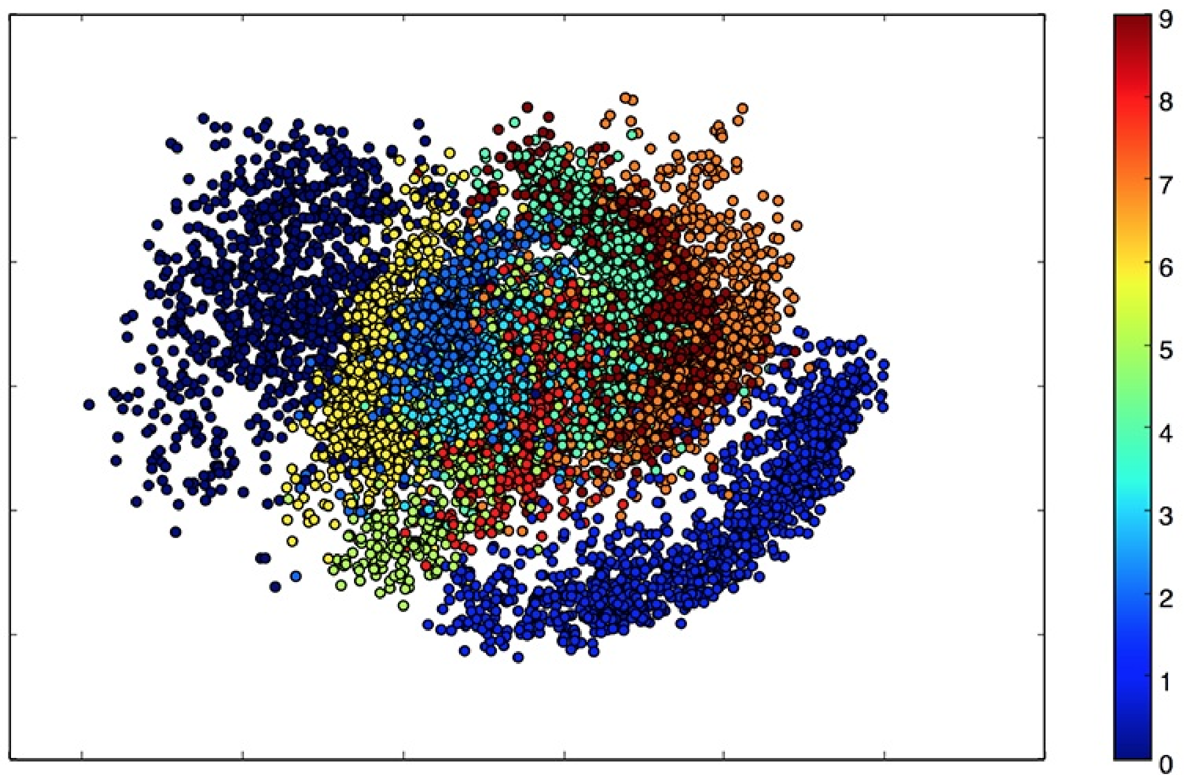
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1
:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:]))
)

vae.fit(x_train, x_train,
        shuffle=True,
        nb_epoch=nb_epoch,
        batch_size=batch_size,
        validation_data=(x_test, x_test))
```

因为我们的隐空间只有两维，所以我们可以可视化一下。我们来看看2D平面中不同类的近邻分布：

```
x_test_encoded = encoder.predict(x_test, batch_size=batch_size)
plt.figure(figsize=(6, 6))
plt.scatter(x_test_encoded[:, 0], x_test_encoded[:, 1], c=y_test
)
plt.colorbar()
plt.show()
```



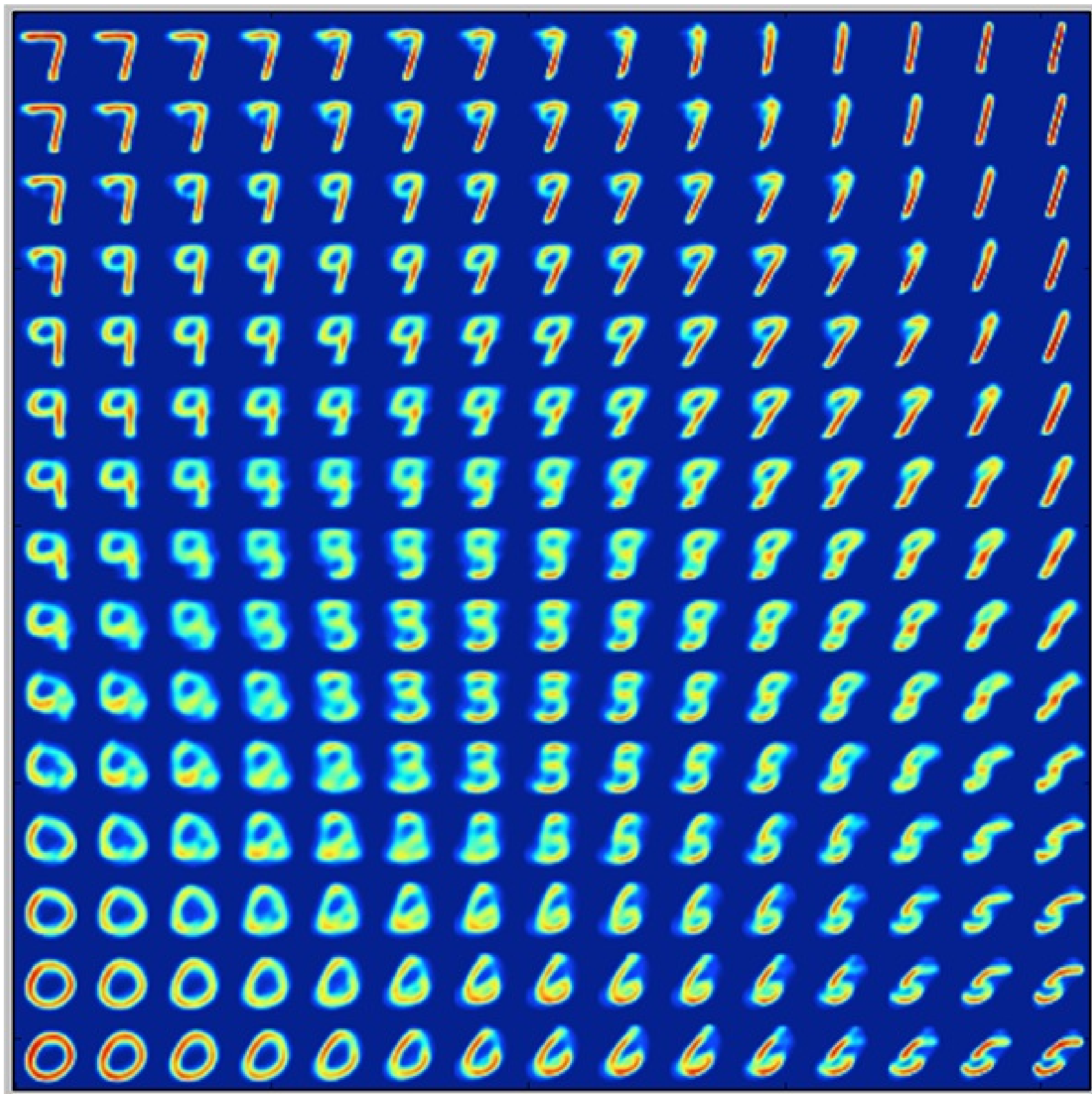
上图每种颜色代表一个数字，相近聚类的数字代表他们在结构上相似。

因为变分编码器是一个生成模型，我们可以用它来生成新数字。我们可以从隐平面上采样一些点，然后生成对应的显变量，即MNIST的数字：

```
# display a 2D manifold of the digits
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# we will sample n points within [-15, 15] standard deviations
grid_x = np.linspace(-15, 15, n)
grid_y = np.linspace(-15, 15, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]]) * epsilon_std
        x_decoded = generator.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```



OK这就是本文的全部，如果你觉得本文还可以增加点别的主题，可以在Twitter上@fchollet

参考文献

- [1] Why does unsupervised pre-training help deep learning?
- [2] Batch normalization: Accelerating deep network training by reducing internal covariate shift.
- [3] Deep Residual Learning for Image Recognition
- [4] Auto-Encoding Variational Bayes

面向小数据集构建图像分类模型

文章信息

本文地址：http://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

本文作者：Francois Chollet

概述

在本文中，我们将提供一些面向小数据集（几百张到几千张图片）构造高效、实用的图像分类器的方法。

本文将探讨如下几种方法：

- 从图片中直接训练一个小网络（作为基准方法）
- 利用预训练网络的bottleneck（瓶颈）特征
- fine-tune预训练网络的高层

本文需要使用的Keras模块有：

- `fit_generator`：用于从Python生成器中训练网络
- `ImageDataGenerator`：用于实时数据提升
- 层参数冻结和模型fine-tune

配置情况

我们的实验基于下面的配置

- 2000张训练图片构成的数据集，一共两个类别，每类1000张
- 安装有Keras，SciPy，PIL的机器，如果有NVIDIA GPU那就更好了，但因为我们面对的是小数据集，没有也可以。

- 数据集按照下面的形式存放

```
data/
    train/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
        cats/
            cat001/jpg
            cat002.jpg
            ...
    validation/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
        cats/
            cat001/jpg
            cat002.jpg
            ...
```

这份数据集来源于Kaggle，原数据集有12500只猫和12500只狗，我们只取了各个类的前1000张图片。另外我们还从各个类中取了400张额外图片用于测试。

下面是数据集的一些示例图片，图片的数量非常少，这对于图像分类来说是个大麻烦。但现实是，很多真实世界图片获取是很困难的，我们能得到的样本数目确实很有限（比如医学图像，每张正样本都意味着一个承受痛苦的病人:()。对数据科学家而言，我们应该有能够榨取少量数据的全部价值的能力，而不是简单的伸手要更多的数据。



在Kaggle的猫狗大战竞赛种，参赛者通过使用现代的深度学习技术达到了98%的正确率，我们只使用了全部数据的8%，因此这个问题对我们来说更难。

针对小数据集的深度学习

我经常听到的一种说法是，深度学习只有在你拥有海量数据时才有意义。虽然这种说法并不是完全不对，但却具有较强的误导性。当然，深度学习强调从数据中自动学习特征的能力，没有足够的训练样本，这几乎是不可能的。尤其是当输入的数据维度很高（如图片）时。然而，卷积神经网络作为深度学习的支柱，被设计为针对“感知”问题最好的模型之一（如图像分类问题），即使只有很少的数据，网络也能把特征学的不错。针对小数据集的神经网络依然能够得到合理的结果，并不需要任何手工的特征工程。一言以蔽之，卷积神经网络大法好！

另一方面，深度学习模型天然就具有可重用的特性：比方说，你可以把一个在大规模数据上训练好的图像分类或语音识别的模型重用在另一个很不一样的问题上，而只需要做有限的一点改动。尤其在计算机视觉领域，许多预训练的模型现在都被公开下载，并被重用在其他问题上以提升在小数据集上的性能。

数据预处理与数据提升

为了尽量利用我们有限的训练数据，我们将通过一系列随机变换堆数据进行提升，这样我们的模型将看不到任何两张完全相同的图片，这有利于我们抑制过拟合，使得模型的泛化能力更好。

在Keras中，这个步骤可以通过 `keras.preprocessing.image.ImageGenerator` 来实现，这个类使你可以：

- 在训练过程中，设置要施行的随机变换
- 通过 `.flow` 或 `.flow_from_directory(directory)` 方法实例化一个针对图像batch的生成器，这些生成器可以被用作keras模型相关方法的输入，如 `fit_generator` ，`evaluate_generator` 和 `predict_generator`

现在让我们看个例子：

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

上面显示的只是一部分选项，请阅读文档的相关部分来查看全部可用的选项。我们来快速的浏览一下这些选项的含义：

- `rotation_range` 是一个0~180的度数，用来指定随机选择图片的角度。
- `width_shift` 和 `height_shift` 用来指定水平和竖直方向随机移动的程度，这是两个0~1之间的比例。
- `rescale` 值将在执行其他处理前乘到整个图像上，我们的图像在RGB通道都是0~255的整数，这样的操作可能使图像的值过高或过低，所以我们将这个值定为0~1之间的数。
- `shear_range` 是用来进行剪切变换的程度，参考剪切变换
- `zoom_range` 用来进行随机的放大
- `horizontal_flip` 随机的对图片进行水平翻转，这个参数适用于水平翻转不影响图片语义的时候

- `fill_mode` 用来指定当需要进行像素填充，如旋转，水平和竖直位移时，如何填充新出现的像素

下面我们使用这个工具来生成图片，并将它们保存在一个临时文件夹中，这样我们可以感觉一下数据提升究竟做了什么事。为了使图片能够展示出来，这里没有使用 `rescaling`

```
from keras.preprocessing.image import ImageDataGenerator, array_
to_img, img_to_array, load_img

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

img = load_img('data/train/cats/cat.0.jpg')  # this is a PIL ima
ge
x = img_to_array(img)  # this is a Numpy array with shape (3, 15
0, 150)
x = x.reshape((1,) + x.shape)  # this is a Numpy array with shap
e (1, 3, 150, 150)

# the .flow() command below generates batches of randomly transf
ormed images
# and saves the results to the `preview/` directory
i = 0
for batch in datagen.flow(x, batch_size=1,
                          save_to_dir='preview', save_prefix='ca
t', save_format='jpeg'):
    i += 1
    if i > 20:
        break  # otherwise the generator would loop indefinitely
```

下面是一张图片被提升以后得到的多个结果：



在小数据集上训练神经网络：40行代码达到80%的准确率

进行图像分类的正确工具是卷积网络，所以我们来试试用卷积神经网络搭建一个初级的模型。因为我们的样本数很少，所以我们应该对过拟合的问题多加注意。当一个模型从很少的样本中学习到不能推广到新数据的模式时，我们称为出现了过拟合的问题。过拟合发生时，模型试图使用不相关的特征来进行预测。例如，你有三张伐木工人的照片，有三张水手的照片。六张照片中只有一个伐木工人戴了帽子，如果你认为戴帽子是能将伐木工人与水手区别开的特征，那么此时你就是一个差劲的分类器。

数据提升是对抗过拟合问题的一个武器，但还不够，因为提升过的数据仍然是高度相关的。对抗过拟合的你应该主要关注的是模型的“熵容量”——模型允许存储的信息量。能够存储更多信息的模型能够利用更多的特征取得更好的性能，但也有存储不相关特征的风险。另一方面，只能存储少量信息的模型会将存储的特征主要集中在真正相关的特征上，并有更好的泛化性能。

有很多不同的方法来调整模型的“熵容量”，常见的一种选择是调整模型的参数数目，即模型的层数和每层的规模。另一种方法是对权重进行正则化约束，如L1或L2.这种约束会使模型的权重偏向较小的值。

在我们的模型里，我们使用了很小的卷积网络，只有很少的几层，每层的滤波器数目也不多。再加上数据提升和Dropout，就差不多了。Dropout通过防止一层看到两次完全一样的模式来防止过拟合，相当于也是一种数据提升的方法。（你可以说dropout和数据提升都在随机扰乱数据的相关性）

下面展示的代码是我们的第一个模型，一个很简单的3层卷积加上ReLU激活函数，再接max-pooling层。这个结构和Yann LeCun在1990年发布的图像分类器很相似（除了ReLU）

这个实验的全部代码在这里

```
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense

model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(3, 150, 150)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# the model so far outputs 3D feature maps (height, width, features)
```

然后我们接了两个全连接网络，并以单个神经元和sigmoid激活结束模型。这种选择会产生二分类的结果，与这种配置相适应，我们使用 `binary_crossentropy` 作为损失函数。

```
model.add(Flatten())  # this converts our 3D feature maps to 1D feature vectors
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

然后我们开始准备数据，使用 `.flow_from_directory()` 来从我们的jpgs图片中直接产生数据和标签。

```
# this is the augmentation configuration we will use for training

train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

# this is the augmentation configuration we will use for testing:

# only rescaling
test_datagen = ImageDataGenerator(rescale=1./255)

# this is a generator that will read pictures found in
# subfolers of 'data/train', and indefinitely generate
# batches of augmented image data
train_generator = train_datagen.flow_from_directory(
        'data/train',  # this is the target directory
        target_size=(150, 150),  # all images will be resized to
 150x150
        batch_size=32,
        class_mode='binary')  # since we use binary_crossentropy
 loss, we need binary labels

# this is a similar generator, for validation data
validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')
```

然后我们可以用这个生成器来训练网络了，在GPU上每个epoch耗时20~30秒，在CPU上耗时300~400秒，所以如果你不是很着急，在CPU上跑这个模型也是完全可以的。

```
model.fit_generator(
        train_generator,
        samples_per_epoch=2000,
        nb_epoch=50,
        validation_data=validation_generator,
        nb_val_samples=800)
model.save_weights('first_try.h5')  # always save your weights a
fter training or during training
```

这个模型在50个epoch后的准确率为79%~81%，别忘了我们只用了8%的数据，也没有花时间来做模型和超参数的优化。在Kaggle中，这个模型已经可以进前100名了（一共215队参与），估计剩下的115队都没有用深度学习:)

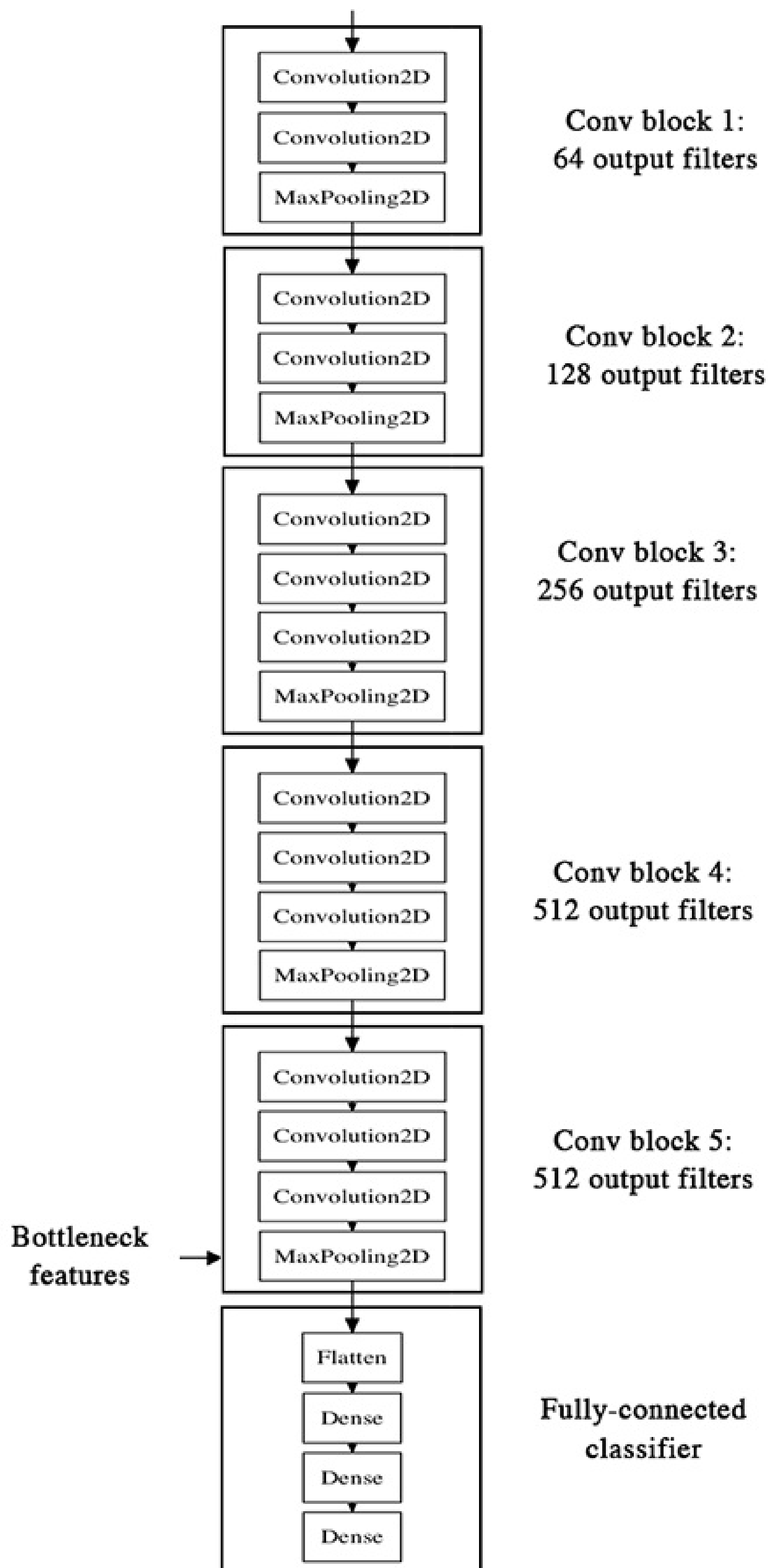
注意这个准确率的变化可能会比较大，因为准确率本来就是一个变化较高的评估参数，而且我们只有800个样本用来测试。比较好的验证方法是使用K折交叉验证，但每轮验证中我们都要训练一个模型。

使用预训练网络的bottleneck特征：一分钟达到90%的正确率

一个稍微讲究一点的办法是，利用在大规模数据集上预训练好的网络。这样的网络在多数的计算机视觉问题上都能取得不错的特征，利用这样的特征可以让我们获得更高的准确率。

我们将使用vgg-16网络，该网络在ImageNet数据集上进行训练，这个模型我们之前提到过了。因为ImageNet数据集包含多种“猫”类和多种“狗”类，这个模型已经能够学习与我们这个数据集相关的特征了。事实上，简单的记录原来网络的输出而不用bottleneck特征就已经足够把我们的问题解决的不错了。不过我们这里讲的方法对其他的类似问题有更好的推广性，包括在ImageNet中没有出现的类别的分类问题。

VGG-16的网络结构如下：


Convolution2D
Convolution2D
MaxPooling2D
Conv block 1:
64 output filters
Convolution2D
Convolution2D
MaxPooling2D
Conv block 2:
128 output filters
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 3:
256 output filters
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 4:
512 output filters
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 5:
512 output filters
Bottleneck
features
Flatten
Dense
Dense
Dense
Fully-connected
classifier

我们的方法是这样的，我们将利用网络的卷积层部分，把全连接以上的部分抛掉。然后在我们的训练集和测试集上跑一遍，将得到的输出（即“bottleneck feature”，网络在全连接之前的最后一层激活的feature map）记录在两个numpy array里。然后我们基于记录下来的特征训练一个全连接网络。

我们将这些特征保存为离线形式，而不是将我们的全连接模型直接加到网络上并冻结之前的层参数进行训练的原因是处于计算效率的考虑。运行VGG网络的代价是非常高昂的，尤其是在CPU上运行，所以我们只想运行一次。这也是我们不进行数据提升的原因。

我们不再赘述如何搭建vgg-16网络了，这件事之前已经说过，在keras的example里也可以找到。但让我们看看如何记录bottleneck特征。

```
generator = datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode=None,  # this means our generator will only yield batches of data, no labels
        shuffle=False)  # our data will be in order, so all first 1000 images will be cats, then 1000 dogs
# the predict_generator method returns the output of a model, given
# a generator that yields batches of numpy data
bottleneck_features_train = model.predict_generator(generator, 2000)
# save the output as a Numpy array
np.save(open('bottleneck_features_train.npy', 'w'), bottleneck_features_train)

generator = datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode=None,
        shuffle=False)
bottleneck_features_validation = model.predict_generator(generator, 800)
np.save(open('bottleneck_features_validation.npy', 'w'), bottleneck_features_validation)
```

记录完毕后我们可以将数据载入，用于训练我们的全连接网络：

```
train_data = np.load(open('bottleneck_features_train.npy'))
# the features were saved in order, so recreating the labels is easy
train_labels = np.array([0] * 1000 + [1] * 1000)

validation_data = np.load(open('bottleneck_features_validation.npy'))
validation_labels = np.array([0] * 400 + [1] * 400)

model = Sequential()
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(train_data, train_labels,
          nb_epoch=50, batch_size=32,
          validation_data=(validation_data, validation_labels))
model.save_weights('bottleneck_fc_model.h5')
```

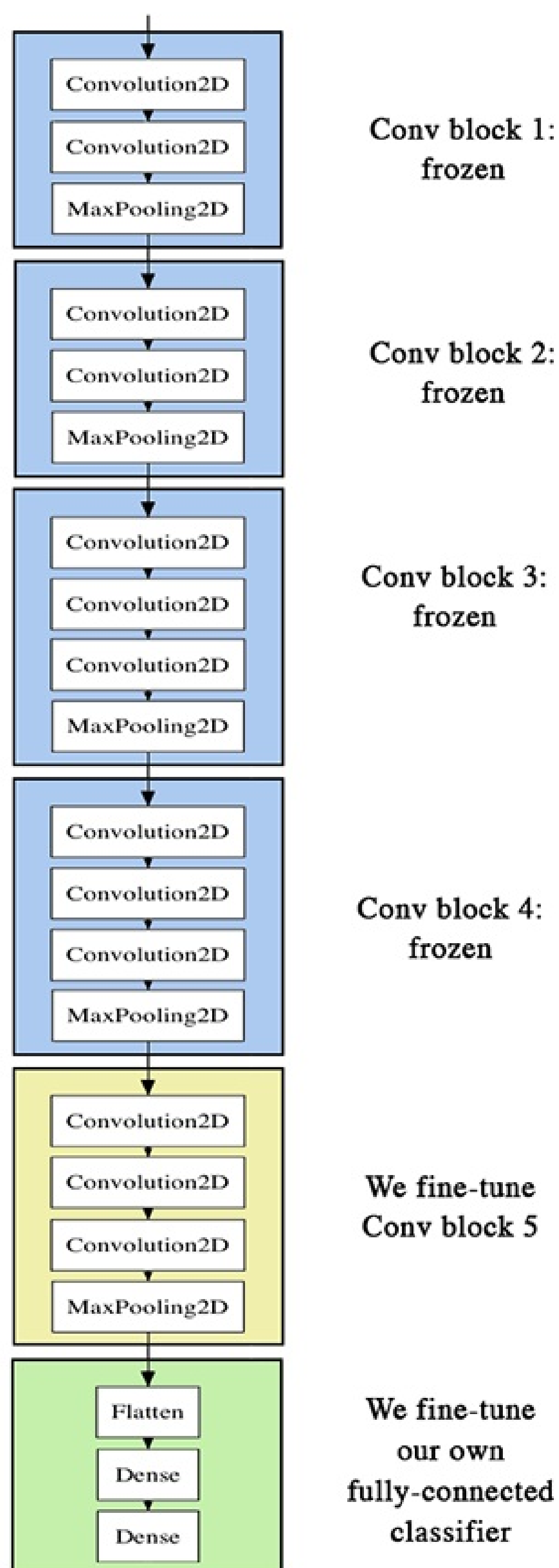
因为特征的size很小，模型在CPU上跑的也会很快，大概1s一个epoch，最后我们的准确率是90%~91%，这么好的结果多半归功于预训练的vgg网络帮助我们提取特征。

在预训练的网络上fine-tune

为了进一步提高之前的结果，我们可以试着fine-tune网络的后面几层。Fine-tune以一个预训练好的网络为基础，在新的数据集上重新训练一小部分权重。在这个实验中，fine-tune分三个步骤

- 搭建vgg-16并载入权重
- 将之前定义的全连接网络加在模型的顶部，并载入权重

- 冻结vgg16网络的一部分参数



注意：

- 为了进行fine-tune,所有的层都应该以训练好的权重为初始值，例如，你不能将随机初始的全连接放在预训练的卷积层之上，这是因为由随机权重产生的大梯度将会破坏卷积层预训练的权重。在我们的情形中，这就是为什么我们首先训练顶层分类器，然后再基于它进行fine-tune的原因
- 我们选择只fine-tune最后的卷积块，而不是整个网络，这是为了防止过拟合。整个网络具有巨大的熵容量，因此具有很高的过拟合倾向。由底层卷积模块学习到的特征更加一般，更加不具有抽象性，因此我们要保持前两个卷积块（学习一般特征）不动，只fine-tune后面的卷积块（学习特别的特征）。
- fine-tune应该在很低的学习率下进行，通常使用SGD优化而不是其他自适应学习率的优化算法，如RMSProp。这是为了保证更新的幅度保持在较低的程度，以免毁坏预训练的特征。

代码如下，首先在初始化好的vgg网络上添加我们预训练好的模型：

```
# build a classifier model to put on top of the convolutional mo
del
top_model = Sequential()
top_model.add(Flatten(input_shape=model.output_shape[1:]))
top_model.add(Dense(256, activation='relu'))
top_model.add(Dropout(0.5))
top_model.add(Dense(1, activation='sigmoid'))

# note that it is necessary to start with a fully-trained
# classifier, including the top classifier,
# in order to successfully do fine-tuning
top_model.load_weights(top_model_weights_path)

# add the model on top of the convolutional base
model.add(top_model)
```

然后将最后一个卷积块前的卷积层参数冻结：

```
# set the first 25 layers (up to the last conv block)
# to non-trainable (weights will not be updated)
for layer in model.layers[:25]:
    layer.trainable = False

# compile the model with a SGD/momentum optimizer
# and a very slow learning rate.
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

然后以很低的学习率进行训练：

```
# prepare data augmentation configuration
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=32,
        class_mode='binary')

# fine-tune the model
model.fit_generator(
        train_generator,
        samples_per_epoch=nb_train_samples,
        nb_epoch=nb_epoch,
        validation_data=validation_generator,
        nb_val_samples=nb_validation_samples)
```

在50个epoch之后该方法的准确率为94%，非常成功

通过下面的方法你可以达到95%以上的正确率：

- 更加强烈的数据提升
- 更加强烈的dropout
- 使用L1和L2正则项（也称为权重衰减）
- fine-tune更多的卷积块（配合更大的正则）

在Keras模型中使用预训练的词向量

文章信息

通过本教程，你可以掌握技能：使用预先训练的词向量和卷积神经网络解决一个文本分类问题 本文代码已上传到Github

本文地址：http://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html

本文作者：Francois Chollet

什么是词向量?

”词向量”（词嵌入）是将一类将词的语义映射到向量空间中去的自然语言处理技术。即将一个词用特定的向量来表示，向量之间的距离（例如，任意两个向量之间的L2范式距离或更常用的余弦距离）一定程度上表征了的词之间的语义关系。由这些向量形成的几何空间被称为一个嵌入空间。

例如，“椰子”和“北极熊”是语义上完全不同的词，所以它们的词向量在一个合理的嵌入空间的距离将会非常遥远。但“厨房”和“晚餐”是相关的话，所以它们的词向量之间的距离会相对小。

理想的情况下，在一个良好的嵌入空间里，从“厨房”向量到“晚餐”向量的“路径”向量会精确地捕捉这两个概念之间的语义关系。在这种情况下，“路径”向量表示的是“发生的地点”，所以你会期望“厨房”向量 - “晚餐"向量（两个词向量的差异）捕捉到“发生的地点”这样的语义关系。基本上，我们应该有向量等式：晚餐 + 发生的地点 = 厨房（至少接近）。如果真的是这样的话，那么我们可以使用这样的关系向量来回答某些问题。例如，应用这种语义关系到一个新的向量，比如“工作”，我们应该得到一个有意义的等式，工作+ 发生的地点 = 办公室，来回答“工作发生在哪里？”。

词向量通过降维技术表征文本数据集中的词的共现信息。方法包括神经网络(“Word2vec”技术)，或矩阵分解。

GloVe 词向量

本文使用GloVe词向量。GloVe 是 "Global Vectors for Word Representation"的缩写，一种基于共现矩阵分解的词向量。本文所使用的GloVe词向量是在2014年的英文维基百科上训练的，有400k个不同的词，每个词用100维向量表示。点此下载(友情提示，词向量文件大小约为822M)

---

20 Newsgroup dataset

本文使用的数据集是著名的"20 Newsgroup dataset"。该数据集共有20种新闻文本数据，我们将实现对该数据集的文本分类任务。数据集的说明和下载请参考这里。

不同类别的新闻包含大量不同的单词，在语义上存在极大的差别，。一些新闻类别如下所示

> comp.sys.ibm.pc.hardware
>
> comp.graphics
>
> comp.os.ms-windows.misc
>
> comp.sys.mac.hardware
>
> comp.windows.x
>
> rec.autos
>
> rec.motorcycles
>
> rec.sport.baseball
>
> rec.sport.hockey

实验方法

以下是我们如何解决分类问题的步骤

- 将所有的新闻样本转化为词索引序列。所谓词索引就是为每一个词依次分配一个整数ID。遍历所有的新闻文本，我们只保留最参见的20,000个词，而且 每个

新闻文本最多保留1000个词。

- 生成一个词向量矩阵。第i列表示词索引为i的词的词向量。
- 将词向量矩阵载入Keras Embedding层，设置该层的权重不可再训练（也就是说在之后的网络训练过程中，词向量不再改变）。
- Keras Embedding层之后连接一个1D的卷积层，并用一个softmax全连接输出新闻类别

数据预处理

我们首先遍历下语料文件下的所有文件夹，获得不同类别的新闻以及对应的类别标签，代码如下所示

```
texts = []  # list of text samples
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)
                f = open(fpath)
                texts.append(f.read())
                f.close()
                labels.append(label_id)

print('Found %s texts.' % len(texts))
```

之后，我们可以新闻样本转化为神经网络训练所用的张量。所用到的Keras库是keras.preprocessing.text.Tokenizer和keras.preprocessing.sequence.pad_sequences。代码如下所示

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

labels = to_categorical(np.asarray(labels))
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
nb_validation_samples = int(VALIDATION_SPLIT * data.shape[0])

x_train = data[:-nb_validation_samples]
y_train = labels[:-nb_validation_samples]
x_val = data[-nb_validation_samples:]
y_val = labels[-nb_validation_samples:]
```

Embedding layer设置

接下来，我们从GloVe文件中解析出每个词和它所对应的词向量，并用字典的方式存储

```
embeddings_index = {}
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

此时，我们可以根据得到的字典生成上文所定义的词向量矩阵

```
embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM)
)
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

现在我们将这个词向量矩阵加载到Embedding层中，注意，我们设置trainable=False使得这个编码层不可再训练。

```
from keras.layers import Embedding

embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
```

一个Embedding层的输入应该是一系列的整数序列，比如一个2D的输入，它的shape值为(samples, indices)，也就是一个samples行，indeces列的矩阵。每一次的batch训练的输入应该被padded成相同大小（尽管Embedding层有能力处理不定

长序列，如果你不指定数列长度这一参数） dim). 所有的序列中的整数都将被对应的词向量矩阵中对应的列（也就是它的词向量）代替,比如序列[1,2]将被序列[词向量[1],词向量[2]]代替。这样，输入一个2D张量后，我们可以得到一个3D张量。

训练1D卷积

最后，我们可以使用一个小型的1D卷积解决这个新闻分类问题。

```
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int3
2')
embedded_sequences = embedding_layer(sequence_input)
x = Conv1D(128, 5, activation='relu')(embedded_sequences)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
x = MaxPooling1D(5)(x)
x = Conv1D(128, 5, activation='relu')(x)
x = MaxPooling1D(35)(x)  # global max pooling
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
preds = Dense(len(labels_index), activation='softmax')(x)

model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['acc'])

# happy learning!
model.fit(x_train, y_train, validation_data=(x_val, y_val),
          nb_epoch=2, batch_size=128)
```

在两次迭代之后，这个模型最后可以达到0.95的分类准确率（4:1分割训练和测试集合）。你可以利用正则方法（例如dropout）或在Embedding层上进行fine-tuning获得更高的准确率。

我们可以做一个对比实验，直接使用Keras自带的Embedding层训练词向量而不用GloVe向量。代码如下所示

```
embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            input_length=MAX_SEQUENCE_LENGTH)
```

两次迭代之后，我们可以得到0.9的准确率。所以使用预训练的词向量作为特征是非常有效的。一般来说，在自然语言处理任务中，当样本数量非常少时，使用预训练的词向量是可行的（实际上，预训练的词向量引入了外部语义信息，往往对模型很有帮助）。

以下部分为译者添加

国内的Rachel-Zhang用sklearn对同样的数据集做过基于传统机器学习算法的实验，请点击这里。 同时Richard Socher等在提出GloVe词向量的那篇论文中指出GloVe词向量比word2vec的性能更好[1]。之后的研究表示word2vec和GloVe其实各有千秋，例如Schnabel等提出了用于测评词向量的各项指标，测评显示 word2vec在大部分测评指标优于GloVe和C&W词向量[2]。本文实现其实可以利用谷歌新闻的word2vec词向量再做一组测评实验。

参考文献

[1]: Pennington J, Socher R, Manning C D. Glove: Global Vectors for Word Representation[C]//EMNLP. 2014, 14: 1532-1543

[2]: Schnabel T, Labutov I, Mimno D, et al. Evaluation methods for unsupervised word embeddings[C]//Proc. of EMNLP. 2015

将Keras作为tensorflow的精简接口

文章信息

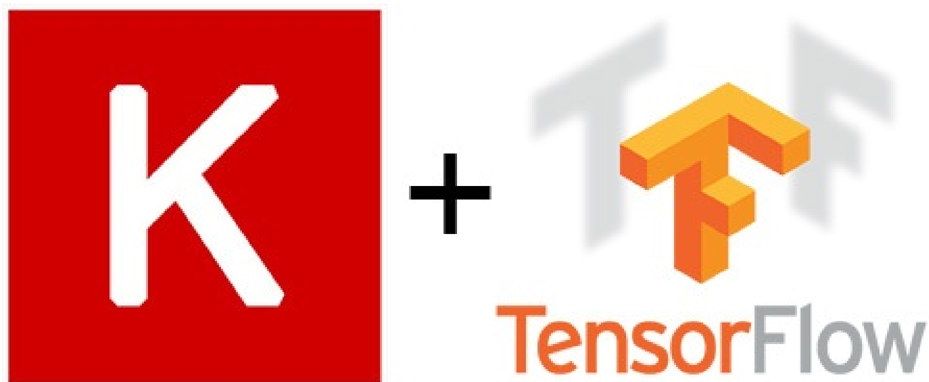
本文地址：https://blog.keras.io/keras-as-a-simplified-interface-to-tensorflow-tutorial.html 本文作者：Francois Chollet

使用Keras作为TensorFlow工作流的一部分

如果Tensorflow是你的首选框架，并且你想找一个简化的、高层的模型定义接口来让自己活的不那么累，那么这篇文章就是给你看的

Keras的层和模型与纯TensorFlow的tensor完全兼容，因此，Keras可以作为TensorFlow的模型定义，甚至可以与其他TensoFlow库协同工作。

注意，本文假定你已经把**Keras**配置为**tensorflow**后端，如果你不懂怎么配置，请查看这里



在tensorflow中调用Keras层

让我们以一个简单的例子开始：MNIST数字分类。我们将以Keras的全连接层堆叠构造一个TensorFlow的分类器，

```
import tensorflow as tf
sess = tf.Session()

from keras import backend as K
K.set_session(sess)
```

然后，我们开始用tensorflow构建模型：

```
# this placeholder will contain our input digits, as flat vectors

img = tf.placeholder(tf.float32, shape=(None, 784))
```

用Keras可以加速模型的定义过程：

```
from keras.layers import Dense

# Keras layers can be called on TensorFlow tensors:
x = Dense(128, activation='relu')(img)  # fully-connected layer
with 128 units and ReLU activation
x = Dense(128, activation='relu')(x)
preds = Dense(10, activation='softmax')(x)  # output layer with
10 units and a softmax activation
```

定义标签的占位符和损失函数：

```
labels = tf.placeholder(tf.float32, shape=(None, 10))

from keras.objectives import categorical_crossentropy
loss = tf.reduce_mean(categorical_crossentropy(labels, preds))
```

然后，我们可以用tensorflow的优化器来训练模型：

```
from tensorflow.examples.tutorials.mnist import input_data
mnist_data = input_data.read_data_sets('MNIST_data', one_hot=True
)

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(los
s)
with sess.as_default():
    for i in range(100):
        batch = mnist_data.train.next_batch(50)
        train_step.run(feed_dict={img: batch[0],
                                  labels: batch[1]})
```

最后我们来评估一下模型性能：

```
from keras.metrics import categorical_accuracy as accuracy

acc_value = accuracy(labels, preds)
with sess.as_default():
    print acc_value.eval(feed_dict={img: mnist_data.test.images,
                                    labels: mnist_data.test.labe
ls})
```

我们只是将Keras作为生成从tensor到tensor的函数（op）的快捷方法而已，优化过程完全采用的原生tensorflow的优化器，而不是Keras优化器，我们压根不需要Keras的Model

关于原生TensorFlow和Keras的优化器的一点注记：虽然有点反直觉，但Keras的优化器要比TensorFlow的优化器快大概5-10%。虽然这种速度的差异基本上没什么差别。

训练和测试行为不同

有些Keras层，如BN，Dropout，在训练和测试过程中的行为不一致，你可以通过打印layer.uses_learning_phase来确定当前层工作在训练模式还是测试模式。

如果你的模型包含这样的层，你需要指定你希望模型工作在什么模式下，通过Keras的backend你可以了解当前的工作模式：

```
from keras import backend as K
print K.learning_phase()
```

向feed_dict中传递1（训练模式）或0（测试模式）即可指定当前工作模式：

```
# train mode
train_step.run(feed_dict={x: batch[0], labels: batch[1], K.learn
ing_phase(): 1})
```

例如，下面代码示范了如何将Dropout层加入刚才的模型中：

```
from keras.layers import Dropout
from keras import backend as K

img = tf.placeholder(tf.float32, shape=(None, 784))
labels = tf.placeholder(tf.float32, shape=(None, 10))

x = Dense(128, activation='relu')(img)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)
x = Dropout(0.5)(x)
preds = Dense(10, activation='softmax')(x)

loss = tf.reduce_mean(categorical_crossentropy(labels, preds))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(los
s)
with sess.as_default():
    for i in range(100):
        batch = mnist_data.train.next_batch(50)
        train_step.run(feed_dict={img: batch[0],
                                  labels: batch[1],
                                  K.learning_phase(): 1})

acc_value = accuracy(labels, preds)
with sess.as_default():
    print acc_value.eval(feed_dict={img: mnist_data.test.images,
                                    labels: mnist_data.test.labe
ls,
                                    K.learning_phase(): 0})
```

与变量名作用域和设备作用域的兼容

Keras的层与模型和tensorflow的命名完全兼容，例如：

```
x = tf.placeholder(tf.float32, shape=(None, 20, 64))
with tf.name_scope('block1'):
    y = LSTM(32, name='mylstm')(x)
```

我们LSTM层的权重将会被命名为block1/mylstm_W_i, block1/mylstm_U, 等.. 类似的，设备的命名也会像你期望的一样工作：

```
with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops / variables in the LSTM layer wil
l live on GPU:0
```

与Graph的作用域兼容

任何在tensorflow的Graph作用域定义的Keras层或模型的所有变量和操作将被生成为该Graph的一个部分，例如，下面的代码将会以你所期望的形式工作

```
from keras.layers import LSTM
import tensorflow as tf

my_graph = tf.Graph()
with my_graph.as_default():
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops / variables in the LSTM layer are
 created as part of our graph
```

与变量作用域兼容

变量共享应通过多次调用同样的Keras层或模型来实现，而不是通过TensorFlow的变量作用域实现。TensorFlow变量作用域将对Keras层或模型没有任何影响。更多Keras权重共享的信息请参考这里

Keras通过重用相同层或模型的对象来完成权值共享，这是一个例子：

```
# instantiate a Keras layer
lstm = LSTM(32)

# instantiate two TF placeholders
x = tf.placeholder(tf.float32, shape=(None, 20, 64))
y = tf.placeholder(tf.float32, shape=(None, 20, 64))

# encode the two tensors with the *same* LSTM weights
x_encoded = lstm(x)
y_encoded = lstm(y)
```

收集可训练权重与状态更新

某些Keras层，如状态RNN和BN层，其内部的更新需要作为训练过程的一步来进行，这些更新被存储在一个tensor tuple里：layer.updates，你应该生成assign操作来使在训练的每一步这些更新能够被运行，这里是例子：

```
from keras.layers import BatchNormalization

layer = BatchNormalization()(x)

update_ops = []
for old_value, new_value in layer.updates:
    update_ops.append(tf.assign(old_value, new_value))
```

注意如果你使用Keras模型，model.updates将与上面的代码作用相同（收集模型中所有更新）

另外，如果你需要显式的收集一个层的可训练权重，你可以通过layer.trainable_weights来实现，对模型而言是model.trainable_weights，它是一个tensorflow变量对象的列表：

```
from keras.layers import Dense

layer = Dense(32)(x)  # instantiate and call a layer
print layer.trainable_weights  # list of TensorFlow Variables
```

这些东西允许你实现你基于TensorFlow优化器实现自己的训练程序

使用Keras模型与TensorFlow协作

将Keras Sequential模型转换到TensorFlow中

假如你已经有一个训练好的Keras模型，如VGG-16，现在你想将它应用在你的TensorFlow工作中，应该怎么办？

首先，注意如果你的预训练权重含有使用Theano训练的卷积层的话，你需要对这些权重的卷积核进行转换，这是因为Theano和TensorFlow对卷积的实现不同，TensorFlow和Caffe实际上实现的是相关性计算。点击这里查看详细示例。

假设你从下面的Keras模型开始，并希望对其进行修改以使得它可以以一个特定的tensorflow张量my_input_tensor为输入，这个tensor可能是一个数据feeder或别的tensorflow模型的输出

```
# this is our initial Keras model
model = Sequential()
first_layer = Dense(32, activation='relu', input_dim=784)
model.add(Dense(10, activation='softmax'))
```

你只需要在实例化该模型后，使用set_input来修改首层的输入，然后将剩下模型搭建于其上：

```
# this is our modified Keras model
model = Sequential()
first_layer = Dense(32, activation='relu', input_dim=784)
first_layer.set_input(my_input_tensor)

# build the rest of the model as before
model.add(first_layer)
model.add(Dense(10, activation='softmax'))
```

在这个阶段，你可以调用model.load_weights(weights_file)来加载预训练的权重

然后，你或许会收集该模型的输出张量：

```
output_tensor = model.output
```

对TensorFlow张量中调用Keras模型

Keras模型与Keras层的行为一致，因此可以被调用于TensorFlow张量上：

```
from keras.models import Sequential

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(10, activation='softmax'))

# this works!
x = tf.placeholder(tf.float32, shape=(None, 784))
y = model(x)
```

注意，调用模型时你同时使用了模型的结构与权重，当你在一个tensor上调用模型时，你就在该tensor上创造了一些操作，这些操作重用了已经在模型中出现的TensorFlow变量的对象

多GPU和分布式训练

将Keras模型分散在多个GPU中训练

TensorFlow的设备作用域完全与Keras的层和模型兼容，因此你可以使用它们来将一个图的特定部分放在不同的GPU中训练，这里是一个简单的例子：

```
with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops in the LSTM layer will live on GP
U:0

with tf.device('/gpu:1'):
    x = tf.placeholder(tf.float32, shape=(None, 20, 64))
    y = LSTM(32)(x)  # all ops in the LSTM layer will live on GP
U:1
```

注意，由LSTM层创建的变量将不会生存在GPU上，不管TensorFlow变量在哪里创建，它们总是生存在CPU上，TensorFlow将隐含的处理设备之间的转换

如果你想在多个GPU上训练同一个模型的多个副本，并在多个副本中进行权重共享，首先你应该在一个设备作用域下实例化你的模型或层，然后在不同GPU设备的作用域下多次调用该模型实例，如：

```
with tf.device('/cpu:0'):
    x = tf.placeholder(tf.float32, shape=(None, 784))

    # shared model living on CPU:0
    # it won't actually be run during training; it acts as an op
 template
    # and as a repository for shared variables
    model = Sequential()
    model.add(Dense(32, activation='relu', input_dim=784))
    model.add(Dense(10, activation='softmax'))

# replica 0
with tf.device('/gpu:0'):
    output_0 = model(x)  # all ops in the replica will live on G
PU:0

# replica 1
with tf.device('/gpu:1'):
    output_1 = model(x)  # all ops in the replica will live on G
PU:1

# merge outputs on CPU
with tf.device('/cpu:0'):
    preds = 0.5 * (output_0 + output_1)

# we only run the `preds` tensor, so that only the two
# replicas on GPU get run (plus the merge op on CPU)
output_value = sess.run([preds], feed_dict={x: data})
```

分布式训练

通过注册Keras会话到一个集群上，你可以简单的实现分布式训练：

```
server = tf.train.Server.create_local_server()
sess = tf.Session(server.target)

from keras import backend as K
K.set_session(sess)
```

关于TensorFlow进行分布式训练的配置信息，请参考这里

使用TensorFlow-serving导出模型

TensorFlow-Serving是由Google开发的用于将TensoFlow模型部署于生产环境的工具

任何Keras模型都可以被TensorFlow-serving所导出（只要它只含有一个输入和一个输出，这是TF-serving的限制），不管它是否作为TensroFlow工作流的一部分。事实上你甚至可以使用Theano训练你的Keras模型，然后将其切换到tensorflow后端，然后导出模型

如果你的graph使用了Keras的learning phase（在训练和测试中行为不同），你首先要做的事就是在graph中硬编码你的工作模式（设为0，即测试模式），该工作通过1）使用Keras的后端注册一个learning phase常量，2）重新构建模型，来完成。

这里是实践中的示范：

```
from keras import backend as K

K.set_learning_phase(0)  # all new operations will be in test mo
de from now on

# serialize the model and get its weights, for quick re-building
config = previous_model.get_config()
weights = previous_model.get_weights()

# re-build a model where the learning phase is now hard-coded to
 0
from keras.models import model_from_config
new_model = model_from_config(config)
new_model.set_weights(weights)
```

现在，我们可使用Tensorflow-serving来导出模型，按照官方教程的指导：

```
from tensorflow_serving.session_bundle import exporter

export_path = ... # where to save the exported graph
export_version = ... # version number (integer)

saver = tf.train.Saver(sharded=True)
model_exporter = exporter.Exporter(saver)
signature = exporter.classification_signature(input_tensor=model
.input,
                                              scores_tensor=mode
l.output)
model_exporter.init(sess.graph.as_graph_def(),
                    default_graph_signature=signature)
model_exporter.export(export_path, tf.constant(export_version),
sess)
```

如想看到包含本教程的新主题，请看我的Twitter

致谢

本项目由以下贡献者贡献：

文档贡献

贡献者	页面	章节	类型
Bigmoyan	keras.io的全部正文	-	翻译
Bigmoyan	深度学习与Keras	CNN眼中的世界	翻译
Bigmoyan	深度学习与Keras	花式自动编码器	翻译
Bigmoyan	快速开始	一些基本概念	编写
SCP-173	快速开始	Keras安装和配置指南(Linux)	编写
SCP-173	快速开始	Keras安装和配置指南(Windows)	编写
Bigmoyan	深度学习与Keras	面向小数据集构建图像分类模型	翻译
leo-nlp	深度学习与Keras	在Keras模型中使用预训练的词向量	翻译
zhourunlai	工具	I/O工具	翻译

Tips

贡献者	页面
Bigmoyan	Tips处标注
3rduncle	Tips处标注
白菜	Tips处标记
我是小将	Tips处标记
zhourunlai	可视化

Keras陷阱提示

贡献者	页面
Bigmoyan	tf与th卷积核陷阱
Bigmoyan	向BN层载入权重陷阱
Yin	validation_spilit与shuffle陷阱
Hui Liu	merge与Merge的区别

Reviewers

Reviewer	页面	章节
白菜	快速开始泛型模型	共享层
白菜	常用数据库	IMDB影评倾向分类
doudou	关于模型	关于Keras模型
doudou	常用层	Dense层
艾子	常用层	Merge层
NUDT-小超人、、	快速开始Sequential模型	Merge层
毛毛熊	常用数据库	cifar-10
迷川浩浩	回调函数	callback
tadakey	一些基本概念	张量
方渺渺	递归层	recurrent层
leo-nlp	快速开始泛型模型	共享层
单车	常用层	Masking层
张涛	快速开始泛型模型	多输入和多输出模型
白菜	FAQ	如何观察中间层的输出
毒液	文本预处理	one-hot
毒液	回调函数	EarlyStopping
毒液	目标函数	可用的目标函数
毛毛熊	正则项	缩写
木子天一	局部连接层LocallyConnceted	LocallyConnected2D层
QiaXi	Pooling层	GlobalMax/GlobalAve
shawn	Callback	ModelCheckpoint
smallYoki	快速开始	泛型模型

示例程序

- 虚位以待